Explotación de aceleradores y hardware gráfico de forma amigable

Nelson Martín Vieites y Jorge Collado García

GRADO EN INGENIERÍA DE COMPUTADORES

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE GRADO EN INGENIERÍA DE COMPUTADORES

Madrid, 20 de junio de 2014

Directores:

Guillermo Botella Juan y Carlos García Sánchez

Autorización de difusión y utilización

Nosotros Nelson Martín Vieites y Jorge Collado García autores de este documento autorizamos a la Universidad Complutense de Madrid a la difusión y utilización del mismo.

Nelson Martín Vieites **Jorge Collado García**

DNI: 11854940-G **DNI: 505549-E**

Madrid, 20 de junio de 2014

A mi hermano Borja, ya que sin su ayuda no hubiera conseguido todo lo que he logrado estos años y porque él es uno de los motivos por los que sé que debo seguir superándome. A mis padres Manuela y Javier por haber creído siempre en mí y conseguir que finalmente yo también lo hiciera. A mi novia Sara por apoyarme en todo momento y darme ese último empujón cuando no me creía capaz de algo

Nelson

A mis "compis" de biblioteca, esos con los que a los cinco minutos de sentarte te ibas a la calle a hacer nada. A aquellos que te obligaban a estudiar por las noches, bajo amenaza. A todos ellos y a mi familia, que gracias a ellos estoy hoy donde estoy

Jorge

Agradecimientos

Quisiéramos mostrar nuestro más sincero agradecimiento a nuestros tutores de proyecto Guillermo Botella Juan y Carlos García Sánchez, ya que sin ellos hubiera sido imposible la realización del mismo. Nos han apoyado y motivado desde el primer momento consiguiendo que nos interesáramos cada día más por la temática del proyecto, al igual que han estado siempre a nuestra disposición para todo aquello que hemos necesitado.

Por supuesto agradecer a nuestras familias, parejas y amigos el apoyo diario que nos han dado durante todos estos años hasta llegar aquí.

Índice

Índice de figuras/tablas

Resumen

Los algoritmos de flujo óptico tienen un alto coste computacional, pero las operaciones que conllevan también muestran un alto grado de paralelismo. Estas dos cualidades convierten a este tipo de algoritmos en buenos candidatos para mejorar su rendimiento en aceleradores y hardware gráfico. El problema que lleva consigo el uso de estos aceleradores para los programadores es la necesidad de conocer su arquitectura, además de lenguajes de programación específicos; siendo muy costosa la tarea de migrar el código para su utilización en este tipo de hardware. La aparición reciente de nuevos paradigmas de programación basados en directivas como OpenMP y OpenACC resuelve dicho problema, ya que con un pequeño porcentaje de modificaciones en el código original (entorno al 5-7%) los algoritmos pueden ser acelerados; pudiéndose considerar un buen balance el obtenido entre el esfuerzo de codificación y rendimiento computacional. En este proyecto se estudiarán los beneficios antes comentados en una implementación del algoritmo de flujo óptico Lucas&Kanade. Para ello se paralelizará con OpenMP sobre una CPU multicore y posteriormente en GPU's mediante OpenACC.

Palabras clave: Aceleradores, CPU, Directivas, Flujo óptico, GPU's, Lucas&Kanade, OpenACC, OpenMP.

Abstract

Optical flow algorithms require a great amount of computational resources, but the operations involving also show high degree of parallelism. Both features make this type of algorithms suitable candidates to improve their performance in accelerators and graphic hardware. For programmers the problem associated with the use of these accelerators is the necessity of knowing their architecture, as well as specific programming languages; being very hard the task of migrating code to its usage on this type of hardware. The recent emergence of new programming paradigms, based on directives as OpenMP and OpenACC solves the problem, because with a small percentage of changes in the original code (with around 5-7%) algorithms can be accelerated; being a good balance the one obtained between the coding effort and computational performance. In this project we will study the previously mentioned benefits in an implementation of the Lucas & Kanade optical flow algorithm. So, it will be parallelized with OpenMP on a multicore CPU, and then on GPU's by Open ACC.

Keywords: Accelerators, CPU, Directives, Optical Flow, GPU's, Lucas&Kanade, OpenACC, OpenMP.

Capítulo 1. Introducción

1.1 Estimación de movimiento y flujo óptico.

El campo de la visión por computador es un subcampo de la inteligencia artificial que se encarga de procesar, analizar y adquirir información de imágenes, con el objetivo de generar información numérica o simbólica. La estimación de movimiento es uno de los problemas que más interés suscitan en este campo, debido a las diversas implicaciones que esta tiene en el mundo real. Por citar algunos ejemplos podemos nombrar la videovigilancia, control de tráfico, robótica e incluso aplicaciones médicas.

El objetivo de la estimación de movimiento es obtener el campo de vectores que representan el movimiento producido entre dos frames (fotograma perteneciente a una secuencia de imágenes) consecutivos en una secuencia de vídeo. Podemos definir el flujo óptico como la velocidad provocada por los cambios de intensidad en una imagen. En las imágenes digitales hay una pérdida de información con respecto al movimiento real que puede provocar, por ejemplo, que en situaciones con cambios de iluminación bruscos el movimiento real y el estimado no coincidan.

1.1.1 Estímulos.

Los estímulos son los datos de entrada utilizados para probar sobre los algoritmos que calculan el flujo óptico; siendo los estímulos sintéticos unos buenos candidatos, ya que al no ser de origen natural sino que han sido creados pensando en ser probados en este tipo de algoritmos, se conoce de antemano mucha información útil. Se conocen exactamente las direcciones del movimiento en dos dimensiones de la imagen, así como otras particularidades que se puedan tener en cuenta a la hora de medir el comportamiento del algoritmo.

El campo de vectores reales que representan el movimiento sobre el estímulo es conocido como “ground truth” y será utilizado junto con algunas de las métricas de error explicadas más adelante para comprobar la precisión de una posible implementación de estas técnicas de estimación de movimiento. Hay que tener en cuenta que este tipo de estímulos tienen condiciones ideales (no hay transparencias, sombras o distintos aspectos que se encuentran en imágenes del mundo real) por lo que son útiles a la hora de probar la precisión de nuestro algoritmo, sin embargo, debemos tener en cuenta que es una medida optimista, y por tanto, es probable que en situaciones reales no obtuviéramos los mismos resultados.

A continuación, describiremos brevemente algunos de los estímulos más conocidos y que hemos utilizado en este proyecto:

Las translaciones de ondas sinusoidales en distintas frecuencias y direcciones permiten apreciar los patrones de movimiento.

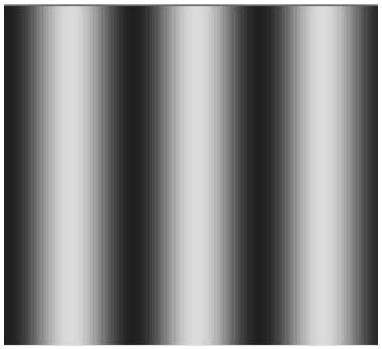

Figura 1: Estímulo sintético que simula la translación de un seno.

Los estímulos Translating Tree y Diverging Tree[1] (David Fleet, Universidad de Toronto) representan respectivamente el movimiento de translación horizontal de un árbol y el efecto de zoom de una cámara. Ambos estímulos tienen unas dimensiones de 150x150 pixeles. En el caso del "Diverging Tree" se tiene un rango de velocidades prácticamente nulo en el centro de la imagen que va creciendo a medida que se aproxima a los extremos, simulando así el efecto de zoom antes mencionado. Para el "Translating tree" el rango de velocidades va variando a lo largo del eje x, siendo diferentes en el borde izquierdo y derecho.

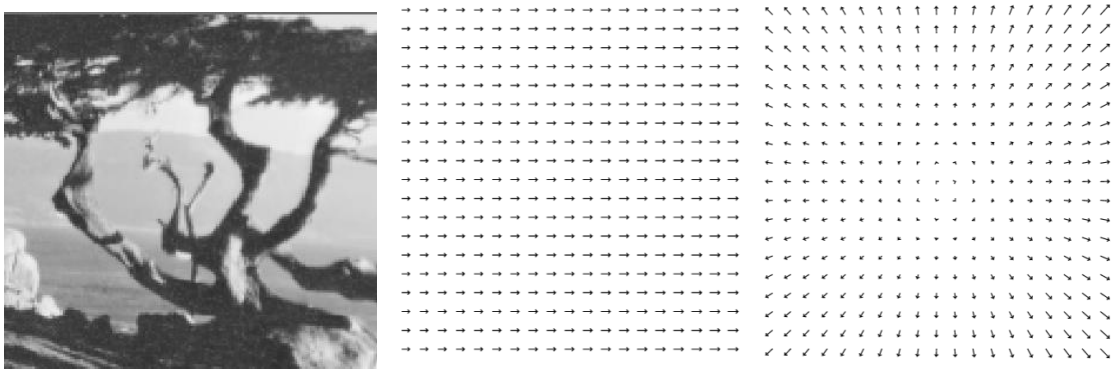

Figura 2: Textura de los estímulos Translating tree y Diverging tree con sus respectivos ground truth.

.

[1] Estímulos disponibles en: http://www.csd.uwo.ca/faculty/barron/FTP/TESTDATA/TREE_DATA/

Un estímulo muy conocido es el de la secuencia de Yosemite[2] (Lynn Quam, Stanford Research Institute), siendo este más complejo que los anteriores, representa el movimiento de una cascada en el parque nacional de Yosemite en Estados Unidos. Dependiendo de la zona de la imagen, encontramos movimiento en distintas direcciones y diferentes velocidades. La existencia de *aliasing*[3] en la parte inferior de la imagen, dificulta las mediciones en muchos algoritmos.

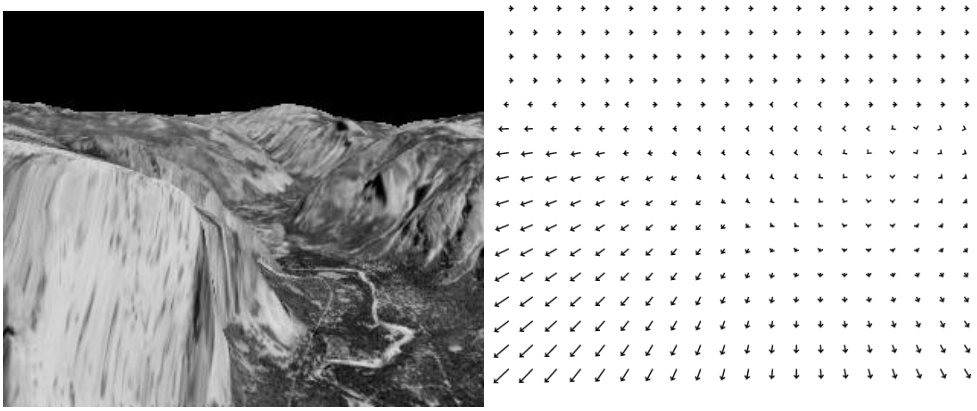

Figura 3: Textura del estímulo Yosemite y flujo óptico real.

1.1.2 Métricas.

Para medir la precisión de la implementación realizada se necesitan métricas de error que permitan comprobar la variación entre el movimiento real del estímulo y el movimiento estimado.

Una de las métricas más utilizadas para flujo óptico es la de John Barron [1]. Esta es una medida de error angular. La velocidad puede describirse como el desplazamiento por unidad de tiempo v = (u, v) pixels/frame o como un vector dirección espacio-temporal (u, v, 1) en unidades (pixel, pixel, frame). Cuando la velocidad es vista como orientación espacio tiempo, se puede medir el error como la desviación angular desde la orientación espacio-temporal correcta. Por lo tanto, deja la velocidad representada como un vector 3D unitario que incluye módulo y fase en un único valor reduciendo el error para velocidades pequeñas. Este vector se representa en la ecuación (1).

[2] Disponible en: http://cs.brown.edu/~black/images.html

[3] En computación gráfica, el aliasing es el artefacto gráfico característico que hace que en una pantalla ciertas curvas y líneas inclinadas presenten un efecto visual tipo "sierra" o "escalón". El aliasing ocurre cuando se intenta representar una imagen con curvas y líneas inclinadas en una pantalla, framebuffer o imagen, pero que debido a la resolución finita del sustrato resulta que éste sea incapaz de representar la curva como tal, y por tanto dichas curvas se muestran en pantalla dentadas al estar compuestas por pequeños cuadrados (los píxeles).

$$\vec{v} = \frac{1}{\sqrt{u^2+v^2+1}} (u, v, 1)^{\mathrm{T}} \qquad (1)$$

El error angular entre la velocidad correcta $\vec{v}_c$ y la velocidad estimada $\vec{v}_e$ viene dado en la ecuación (2).

$$\psi_{\mathrm{E}} = arc\ cos\ (\vec{v}_c \cdot \vec{v}_e) \qquad (2)$$

1.2 Clasificación de algoritmos.

Existen tres modelos o categorías de algoritmos y técnicas de flujo óptico que son los modelos de gradiente, de energía y de matching o emparejamiento. Se elegirá uno u otro dependiendo de la aplicación que se les quiera dar y teniendo en cuenta cuál se ajusta mejor al llamado *problema global de la correspondencia*, conformado por los problemas de *aliasing* y *apertura.*

1.2.1 Modelos de gradiente.

Los modelos de gradiente o diferenciales basan su metodología en aplicar derivadas espacio-temporales, sobre las intensidades de los pixeles de la imagen y de esta forma obtener los vectores velocidad que conforman el flujo óptico. Estos vectores velocidad se obtienen a partir de cocientes sobre las derivadas espaciales y temporales calculadas anteriormente. Este enfoque, en líneas generales, proporciona una buena estimación.

El modelo de gradiente tiene un enfoque contrario al de los modelos de energía y matching que serán explicados detalladamente más adelante. El enfoque de estos modelos básicamente utiliza plantillas para obtener un ajuste entre el movimiento y la plantilla. Además estos métodos tienen una gran dependencia del contraste, lo que implica añadir etapas de normalización costosas. También cabe mencionar que la velocidad se calcula en otra etapa extra ya que las plantillas no proporcionan el movimiento.

Por consiguiente, podemos ver que el enfoque de los métodos basados en gradiente tienen un menor número de etapas, puesto que las velocidades son calculadas directamente como un cociente de las derivadas espacio-temporales de cada pixel de la imagen. Esto también tiene el beneficio de que el contraste varía de la misma forma en el numerador y en el denominador, por lo tanto no es necesario una etapa adicional de normalización del contraste para las imágenes.

Como inconveniente a estos métodos encontramos que es neceario un filtrado previo al procesamiento que implica invertir grandes matrices como ocurre con el algoritmo Lucas&Kanade [2, 3].

1.2.2 Modelos de energía.

El mecanismo de los modelos de energía consiste en utilizar filtros orientados espacio-temporales que responden de manera óptima a ciertas velocidades. Este procesado se lleva a cabo con filtros en paralelo activados para un cierto rango de valores. La elección del diseño de los filtros es una de las principales diferencias entre los distintos algoritmos de este modelo.

La estimación de movimiento se plantea como un problema de estimación global bayesiano, se busca maximizar la probabilidad del campo de movimiento observando la intensidad en la siguiente trama. Se utilizan dos funciones de densidad de probabilidad, una es la probabilidad condicional de la intensidad de la imagen observada a partir del campo de movimiento, y la otra la probabilidad a priori de los vectores de movimiento.

Un inconveniente de este modelo es el tiempo que necesitan sus cálculos, ya que se segmenta el campo de movimiento en regiones, procesando pixel a pixel e incluyendo una distribución de velocidades a cada uno.

Esta familia se asemeja al modelo de gradiente en la utilización de filtros espacio-temporales, pero divergen en su utilización. En los modelos de energía los filtros responden a orientaciones espacio-temporales específicas, mientras que en los de gradiente se realiza un cociente con los filtros.

1.2.3 Modelos de matching.

El enfoque de este modelo es uno de los más intuitivos, su metodología se resume en comparar regiones de la imagen entre sucesivos frames, para así obtener el movimiento observando los cambios entre regiones. Este modelo también es conocido como "de emparejamiento" o "Block Matching".

Las regiones en que se divide cada imagen de la secuencia se llaman macrobloques. Se busca el movimiento entre las imágenes comparando estos macrobloques. Se comparan los bloques del frame actual con los del antecesor, deslizando estos sobre una región de pixeles del frame destino. Los cambios infieren la velocidad debido a que ha habido un cambio en el tiempo.

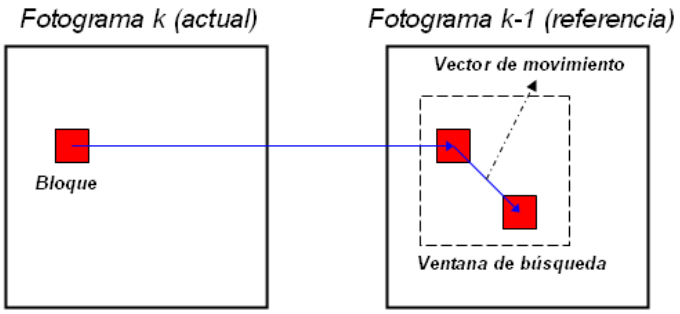

Figura 4: Representación gráfica de un modelo de matching.

Algunos de los criterios de semejanza entre bloques más comunes son, el de Sumatorio de Diferencias Absolutas (SAD), Correlación Cruzada Normalizada (NCC) y Clasificación por Diferencia de Pixel (PDC). Estos criterios se encargan de buscar el bloque más adecuado (mayor similitud) de los que se encuentran en la ventana de búsqueda del frame destino. Si el bloque elegido se encuentra desplazado quiere decir que hay movimiento, y este desplazamiento formará el vector desplazamiento que se asignará por igual a todos los pixeles del bloque. Los bloques coincidentes entre frames no serán exactamente iguales debido al ruido.

El inconveniente de estos algoritmos es la lentitud, esto es debido a una ejecución iterativa y a una búsqueda exhaustiva que requieren muchos recursos. Una imagen de tamaño M tendría una plantilla de tamaño N y una ventana de tamaño L, lo que en orden de complejidad de cómputo daría lugar a MxNxL. Hay variantes de estos algoritmos que tratan de minimizar esta búsqueda exhaustiva FST (Full Search Technique) como TSST (Three Step Search Technique), LOGST (2D Logarithm Search Technique), DS (Diamond Search), CSA (Cross Search Algorithm) y 4SST (Four Step Search Technique).

Como virtud, es destacable su simplicidad, de ahí su amplia aplicación en sectores industriales o en estándares de codificación y compresión de video.

1.3 Algoritmo Lucas&Kanade.

Hay muchos estudios sobre algoritmos de flujo óptico en los cuales sus autores abordan el tema de la precisión de éstos sobre estímulos sintéticos ya que, a priori, el flujo óptico en estímulos reales es desconocido.

El algoritmo de Lucas&Kanade es un clásico en los modelos de gradiente y diversos estudios [3, 4, 5] destacan en él un buen balance entre precisión y eficiencia, requiriendo

unos recursos computaciones abordables. Estos son factores importantes a la hora de decidir que enfoque es más adecuado para implementar un sistema que sea capaz de procesar en tiempo real.

A continuación, describiremos brevemente los cálculos y ecuaciones en los cuales está basado el enfoque de Lucas&Kanade. Se puede encontrar información más detallada en [1, 2, 6].

El algoritmo pertenece a las técnicas de gradiente, las cuales se caracterizan por la búsqueda del gradiente sobre derivadas espaciales y temporales. En el supuesto de valores de iluminación constantes a través del tiempo, la ecuación de gradiente de primer orden se obtiene con la ecuación (3).

$$\nabla_{\mathrm{xy}} I(x,y,t) \cdot \left(v_x, v_y\right) + I_t(x,y,t) = 0 \qquad (3)$$

Esta ecuación solo permite estimar la velocidad en la dirección del gradiente máximo, es decir, en la dirección normal de las superficies de movimiento. Para superar este inconveniente, el modelo construye una estimación de movimiento basada en las derivadas de primer orden de la imagen. Por medio del ajuste de mínimos cuadrados, el modelo extrae la estimación de movimiento bajo la hipótesis de que las velocidades en la vecindad de un pixel central son similares. Esto se describe en la expresión (4),

$$min \sum_{x\in\Omega} W^2(x) \left[I(x,y,t) \cdot \left(v_x, v_y\right) + I_t(x,y,t)\right]^2 \qquad (4)$$

donde W(x) son los pesos que serán asignados a los pixeles de la vecindad espacial Ω, ya que en la práctica, es mejor darle más prioridad a los pixeles situados más cerca del pixel central que se está tratando.

La solución al problema viene dada en la ecuación (5),

$$\vec{v} = \left[A^T W^2 A\right]^{-1} A^T W^2 \vec{b} \qquad (5)$$

donde:

$$A^T W^2 A = \begin{bmatrix} \sum_{x\in\Omega} W^2 I_x^2 & \sum_{x\in\Omega} W^2 I_x I_y \\ \sum_{x\in\Omega} W^2 I_x I_y & \sum_{x\in\Omega} W^2 I_y^2 \end{bmatrix} \qquad (6)$$

$$A^T W^2 \vec{b} = \begin{bmatrix} -\sum_{x\in\Omega} W^2 I_x I_t \\ -\sum_{x\in\Omega} W^2 I_y I_t \end{bmatrix} \qquad (7)$$

Una limitación de este modelo se produce en las situaciones en las que se produce el llamado problema de apertura. En estos casos la matriz de la expresión (6) no se puede invertir, y por lo tanto no se puede obtener una estimación de movimiento en ese punto.

Para concluir la estimación de movimiento entre dos imágenes consecutivas, viene dada la ecuación (5), siendo $\vec{v} = (v_x, v_y)$ el vector velocidad, que será calculado como la multiplicación de la matriz 2x2 de la ecuación (6) invertida y la matriz 2x1 de la ecuación (7).

1.4 Motivación.

El cálculo del flujo óptico es un problema, que como hemos visto hasta ahora tiene un coste computacional considerable. Además por la naturaleza del problema es deseable que la estimación sea lo más precisa posible. Por ello es necesario buscar un equilibrio entre precisión y eficiencia. Con precisión nos referimos a la exactitud de los resultados y con eficiencia a los recursos empleados y tiempo invertido en obtenerlos. Las técnicas más precisas implican mayor requerimiento computacional. Dependiendo de las técnicas utilizadas tendremos implementaciones lentas y precisas o más rápidas y con menos nivel de precisión.

Para el estudio de este proyecto hemos elegido el método de Lucas&Kanade, el cual pertenece a la familia de los modelos de gradiente. Su elección es debido a que ya existen varias implementaciones sobre aceleradores gráficos, algunas de las cuales citaremos a continuación, donde se demuestra la efectividad y la validez de este método que posee un buen compromiso entre precisión y eficiencia, obteniendo valores dentro de lo considerado tiempo real (a partir de 25 fps).

Autor	**fps (Resolución)**	**GPU**	**Año**
Marzat [7]	47 (316x252)	Tesla C870	2009
Duvenhage [8]	30 (512x512)	GTX 285	2010
Ohmura [9]	40 (512x384)	Tesla 1060	2011

Tabla 1: Implementaciones del algoritmo Lucas-Kanade sobre GPUs.

En el trabajo de Marzat se implementa una versión piramidal del Lucas&Kanade sobre la arquitectura CUDA, obteniendo una ganancia en velocidad de 100x sobre la implementación secuencial sobre CPU.

Duvenhage lleva a cabo su implementación sobre GPU utilizando el lenguaje OpenGL Shading Language[4] (GLSL), con el objetivo de aplicarlo en el campo de estabilización de imágenes. Alcanzando en su configuración optima 30 fps sobre resoluciones de 512x512.

El estudio realizado por Ohmura utiliza la estimación de movimiento a través del Lucas&Kanade, para un programa que simula el procesamiento visual del cerebro humano. Esta implementación se lleva a cabo sobre un cluster de GPUs, introduciendo como mejoras la asignación de datos entre las memorias compartida y global de la GPU, además del reparto de convoluciones entre las distintas GPUs para los mismos datos de entrada.

1.5 Objetivos.

El objetivo principal de este trabajo Fin de Grado es evaluar el comportamiento de un algoritmo de estimación de movimiento basado en el método de Lucas&Kanade haciendo uso de los aceleradores gráficos. La decisión de emplear dicha implementación viene motivada por la amplia aceptación de este algoritmo de gradiente entre la comunidad científica.

Debido a las necesidades computaciones del método el uso de aceleradores hardware para completar requisitos de tiempo real ha sido una alternativa considerada con éxito por la comunidad. Sin embargo en la actualidad el uso de aceleradores conlleva la codificación ad-hoc dependiendo del tipo de acelerador considerado. CUDA y OpenCL son los estándares de facto para sistemas basados en procesadores gráficos que son empleados como coprocesadores. El uso de estos interfaces de programación lleva asociado la reescritura completa de la aplicación. Entre las principales desventajas podemos destacar portabilidad limitada de la aplicación desarrollada y la ardua tarea de re-codificación motivada por falta de herramientas.

Bajo este contexto en los últimos años surgen iniciativas que permitan hacer uso de estas tecnologías sin necesidad de reescribir el código fuente o al menos reducir las modificaciones. Entre los resultados con mayor futuro destacamos la programación por medio de directivas que es una solución intermedia entre la re-codificación en CUDA u OpenCL, y una herramienta automática que permita su completa portabilidad. Consideramos que OpenMP[5] por su recorrido en el ámbito de los sistemas con memoria

[4] http://www.opengl.org/documentation/glsl/
[5] http://openmp.org/wp/

compartida es un buen candidato a incrementar sus prestaciones y ampliar su uso a estos sistemas. También podemos destacar la iniciativa que aglutina OpenACC[6] que tiene bastantes visos de asentarse como estándar de programación por medio de directivas para aceleradores gráficos.

Los objetivos de este trabajo los podemos resumir en:

- Desarrollar una codificación del algoritmo de Lucas&Kanade en un lenguaje de Alto Nivel como C.

- Estudiar su comportamiento y precisión a la hora de detectar movimiento empleado como casos de prueba una serie de estímulos de entrada ampliamente aceptado por la comunidad.

- Evaluar las partes más costosas del algoritmo que serán candidatas a optimizar.

- Desarrollar las versiones optimizadas del algoritmo haciendo uso de paralelización por medio de directivas. Inicialmente se desarrollará una versión para un sistema multicore desarrollada con OpenMP y posteriormente se desarrollará su versión homónima para aceleradores mediante directivas OpenACC.

- Por último se evaluarán las ganancias en tiempo de ejecución haciendo uso de ambas optimizaciones.

[6] http://www.openacc.org/

Capítulo 2. Hardware asociado y paradigmas de programación

2.1 Unidades de procesamiento gráfico.

2.1.1 Historia.

Las primeras tarjetas gráficas que empezaron a desarrollarse datan de los años sesenta, cuando se utilizaban las impresoras como elemento de visualización. Con la aparición de este nuevo hardware, se comenzaron a utilizar los monitores como elementos de visualización sustituyendo a las citadas impresoras. En estos primitivos aparatos solo era posible visualizar texto, pero con la aparición de chips de carácter gráfico, como el Motorola 6845, se comenzó a dotar a estos novedosos equipos con capacidades gráficas.

Posteriormente, a finales de los años ochenta, se presentaron las primeras tarjetas gráficas conocidas como VGA (Video Graphics Array). Estas tarjetas contaban con una resolución de 640x480 y 256 colores diferentes. Esto supuso el comienzo de la "era gráfica" de los computadores, que pronto daría un giro importante.

En 1995, comenzaron a popularizarse los videojuegos y junto con el auge de las videoconsolas, los requisitos de realismo gráfico supusieron mayores demandas computacionales y no tardaron en aparecer las tarjetas 2D/3D. Este tipo de tarjetas seguían la evolución de VGA, el estándar SVGA, que implementaba alguna función 3D.

A partir del cambio de siglo, los científicos del sector informático comenzaron a utilizar los procesadores gráficos con otros fines que no fuesen los videojuegos. Para acelerar cómputos de aplicaciones científicas/matemáticas empezaron a utilizar las tarjetas gráficas GPU´s. En este ámbito nació un nuevo concepto que fue la utilización de las GPU para aplicaciones de propósito general (General Purpose GPU o GPGPU).

Comenzaron a obtenerse resultados muy buenos en la comparativa de rendimiento respecto a las CPU (por encima de 100x en algunos casos). El problema que se presentó entonces fue la dificultad para programar las GPGPU, siendo necesarias APIs de programación de gráficos como OpenGL para explotar el paralelismo, lo cual limitaba el acceso a la comunidad científica.

Por aquel entonces, la empresa NVIDIA invirtió grandes sumas para conseguir que las GPUs fueran totalmente programables y ofreció transparencia para los desarrolladores con lenguajes como C/C++, Fortran, OpenCL, etc. A consecuencia de estos avances, en 2006, NVIDIA presentó una nueva arquitectura denominada CUDA[10] (Compute Unified Device Architecture), a la vez que la nueva arquitectura gráfica G80 y el modelo de GPU que lo soportaba (GeForce 8800 GTX)[7] Desde entonces, ha ido evolucionando hasta la actual arquitectura NVDIA Kepler[8] El inconveniente de CUDA es que solo funciona en arquitecturas de NVIDIA y por este motivo surgió OpenCL [11] que al contrario que CUDA si permite ser ejecutado en distintas plataformas como CPUs, GPUs o FPGAs.

2.1.2 Estado del arte.

Gracias a la enorme industria del videojuego, desde 2003 las tarjetas gráficas han ido evolucionando para convertirse en potentes aceleradores. Los microprocesadores de las GPUs (Graphic Processed Units) han mantenido el liderazgo en torno al rendimiento de operaciones de punto flotante.

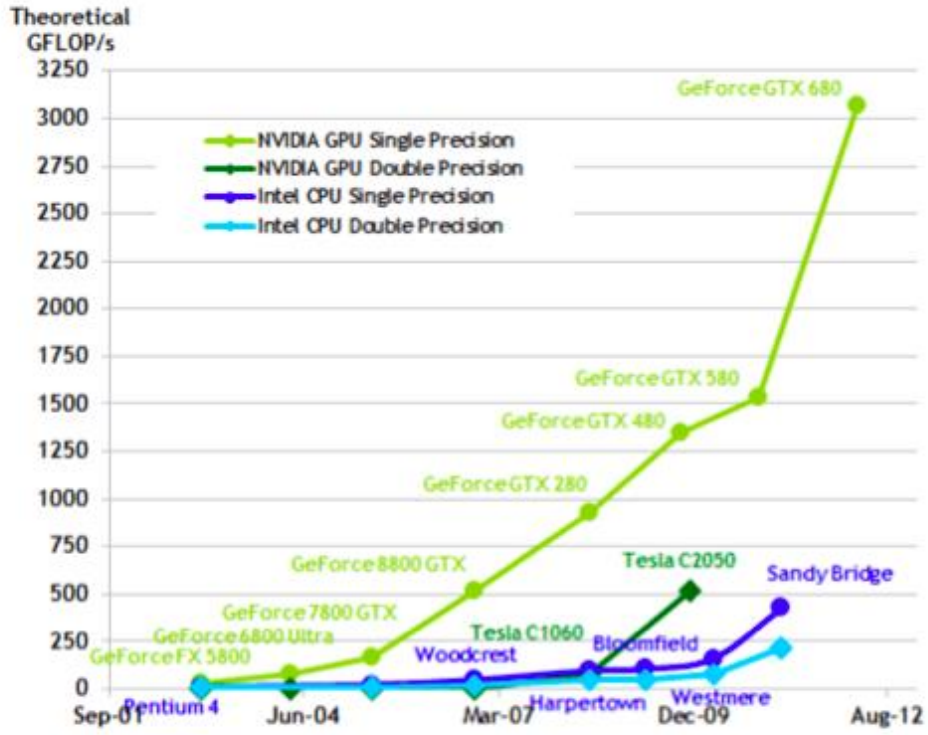

Figura 5: Comparativa de rendimiento en GFLOPS de procesadores y GPUs actuales.

[7] http://www.nvidia.es/page/geforce_8800.html
[8] http://www.nvidia.es/object/nvidia-kepler-es.html

Observamos en dicha gráfica una comparativa del rendimiento en GFLOPS entre los procesadores y las GPUs actuales. Se ve con claridad que desde el 2006, con el lanzamiento de la GeForce 8800 GTX, NVIDIA ha dado un salto de calidad distanciándose del rendimiento de las CPU de propósito general; hasta llegar a la GeForce GTX 680, capaz de procesar hasta tres trillones de operaciones en punto flotante por segundo (3 TFLOPS). La razón de esta diferencia de rendimientos es debido a la especialización por parte de las GPU en computación paralela e intensiva. Están diseñadas de tal manera que se destinan la mayoría de los transistores a la lógica computacional, dotando a la misma de más unidades de computo en vez de más jerarquías de memoria como usan las CPUs. Al ser idóneas para resolver problemas con gran paralelismo de datos se hace una analogía entre los procesadores de las GPU, los cuales siguen un modelo SIMT (Single Instruction Multipe Thread) con los procesadores vectoriales los cuales siguen el modelo SIMD (Single Instruction Multiple Data). Un ejemplo de esta analogía se encuentra en libros de autores como Henessy&Patterson [12].

AMD/ATI y NVIDIA siempre han sido las dos compañías por excelencia para el desarrollo de tarjetas gráficas, siguiendo dos hilos, el comercial y el científico. Actualmente ambas empresas ponen a disposición el conjunto de instrucciones nativas para el cálculo paralelo en las GPUs.

Para competir con la arquitectura creada por NVIDIA, CUDA, en el año 2006 AMD (por aquel entonces no se habían unido AMD y ATI) presenta el repertorio de instrucciones CTM (Close To Metal) para el desarrollo de una programación de propósito general en las tarjetas gráficas de la compañía. Debido al escaso éxito que obtuvo, en 2008 AMD/ATI optó por cancelar el proyecto CTM y se decidieron a apostar por OpenCL. En ese mismo año fue presentado un marco de desarrollo (ATI Stream) que permitia trabajar sobre OpenCL como lenguaje de programación para utilizar las GPUs de la marca como procesadores de propósito general. Por su parte, NVIDIA ofrece desde 2006 algo parecido con su arquitectura CUDA para un propósito similar al de su competencia ATI, siendo éste, además de un marco que permite el uso de OpenCL, un lenguaje específico para el procesado de GPGPU sobre sus propias tarjetas.

El relativo bajo coste de las tarjetas gráficas ha provocado un gran aumento en el desarrollo de las mismas. De esta manera, podemos observar que no solo existen tarjetas gráficas como las descritas anteriormente, sino modelos muy superiores orientados a la computación de altas prestaciones. A modo de ejemplo, existen chips NVIDIA GK110 [13] como el que usa la GPU NVIDIA Tesla K20X, que alcanza un rendimiento máximo de 3.95 TeraFLOPS con 2688 cores y arquitectura Kepler, mientras que el procesador Intel Core i7-3930 [14] solo alcanza 154 GigaFLOPS.

Como se aprecia en la figura que viene a continuación, el uso de este tipo de arquitecturas se está extendiendo cada vez más, encontrando dentro del TOP 5009 de supercomputadores un incremento paulatino de su utilización comparado con los precedentes.

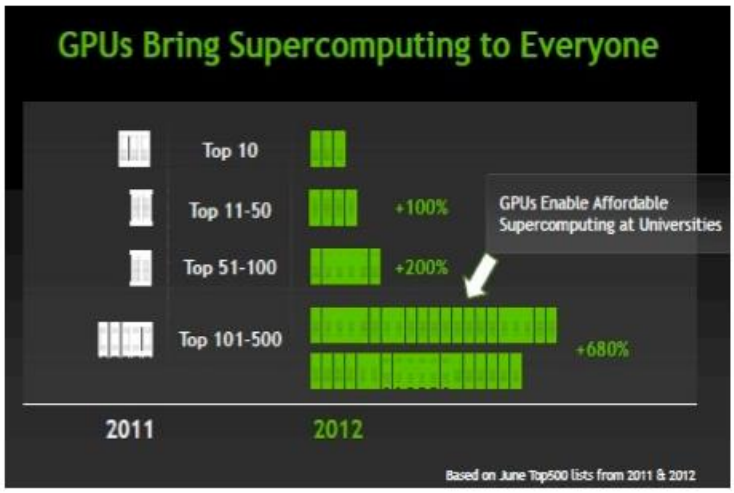

Figura 6: Incremento del uso de GPU en supercomputación.

Si analizamos las características del segundo supercomputador del TOP500 (tabla de 2013), el Cray Titan, se observa que posee 299.008 cores de AMD Opteron y 18.688 aceleradores GPU de NVIDIA Tesla K20.

Como complemento al TOP500, se puede consultar la lista Green50010, cuyo objetivo es el desarrollo de supercomputadores según la eficiencia energética. El objetivo de "buen rendimiento a cualquier precio" ha llevado a la aparición de supercomputadores que consumen grandes cantidades de energía eléctrica. Por ello, desde 2007, la lista Green500 premia el ahorro de estas ingentes cantidades de consumo eléctrico. El Cray Titan antes descrito ocupaba la tercera posición (lista del año 2012), aunque actualmente (lista de noviembre de 2013) ya no se encuentra en el top10 de este ranking. Otro ejemplo de las líneas de investigación actuales que persiguen el bajo consumo sería el proyecto europeo Mont-Blanc11, que tiene como principal objetivo desarrollar un supercomputador energéticamente eficiente con tecnología de bajo consumo. Este proyecto se ha decantado por la plataforma Samsung Exynos para construir su prototipo de supercomputador que integra tecnología de Tablets y Smartphones.

[9] Top500 Supercomputing. http://www.top500.org/, noviembre 2013.
[10] Green500 Supercomputing. http://www.green500.org/, diciembre 2012.
[11] Mont-Blanc Project. http://www.montblanc-project.eu/.

2.1.3 Otras unidades.

Finalmente, cabe destacar la aparición de nuevas formas de configuración CPU–GPU usadas como aceleradores. En este apartado cabe resaltar los nuevos coprocesadores de Intel (los Xeon Phi) y los nuevos procesadores de AMD (A-series).

Los Intel Xeon Phi[12] son tarjetas adicionales con formato PCI que funcionan sinérgicamente con los procesadores Intel Xeon para permitir ganancias considerables de rendimiento para código altamente paralelo (hasta 1,2 TFLOPS). El elevado grado de paralelismo compensa la menor velocidad de cada núcleo ofreciendo un mayor rendimiento adicional para cargas de trabajo con paralelismo.

Mientras que la mayoría de las aplicaciones obtienen el máximo rendimiento con los procesadores Intel Xeon, determinadas aplicaciones que implican paralelismo se benefician considerablemente al utilizar los coprocesadores Intel Xeon Phi. Ejemplos de sectores de la industria con aplicaciones altamente paralelas son: el energético, ciencias biológicas, medicina, meteorología, etc.

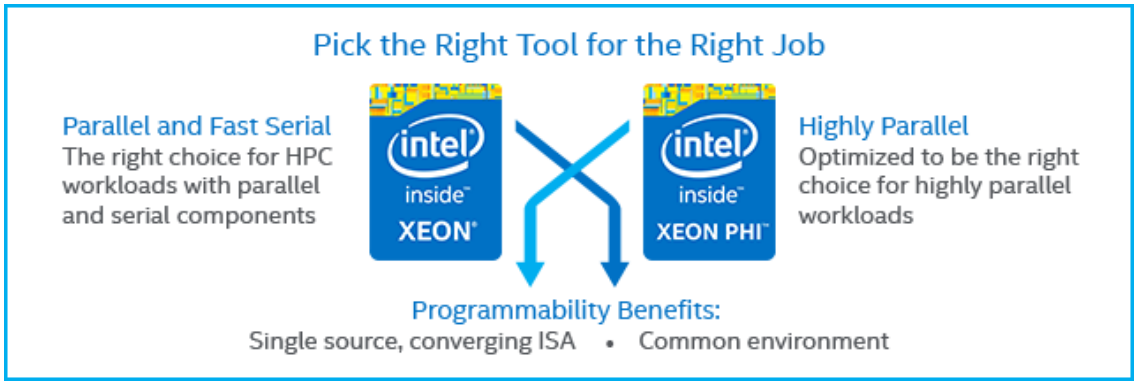

Figura 7: Propósito de los procesadores Intel Xeon y de los coprocesadores Intel Xeon Phi.

Una de las ventajas de estos componentes es que conservan un único código fuente entre los procesadores y los coprocesadores, los desarrolladores optimizan una vez para el paralelismo pero maximizan el rendimiento en el procesador y en el coprocesador. Por otro lado, goza de mucha flexibilidad en la ejecución. A diferencia de una GPU normal, un coprocesador puede alojar un sistema operativo, recibir direcciones IP y soportar estándares como MPI. Además, puede trabajar en múltiples modos de ejecución como son:

[12] http://www.intel.es/content/www/es/es/processors/xeon/xeon-phi-detail.html

- Modo symmetric "simétrico": las cargas de trabajo se comparten entre el procesador anfitrión y el coprocesador.

- Modo native "nativo": la carga de trabajo reside completamente en el coprocesador y actúa fundamentalmente como un nodo informático independiente.

- Modo offload "descarga": la carga de trabajo reside en el procesador anfitrión y partes de esta se envían al coprocesador según sea necesario.

La empresa AMD/ATI está centrada en la fabricación de procesadores con un elevado número de núcleos y con una integración total entre el propio microprocesador y la tarjeta gráfica. Es el primer fabricante que utiliza el concepto de APU en vez del de CPU, es decir, es el primero que integra procesadores gráficos y cores de CPU en el mismo chip. Con esto intentan ofrecer una unidad de procesamiento capaz de trabajar con datos complejos de forma versátil.

En el terreno gráfico, la adquisición de la empresa ATI por parte de AMD ha propiciado que esta última se distancie de su competidor Intel en el mercado de las tarjetas gráficas integradas. Gracias a que las tarjetas gráficas están pensadas para trabajar con datos en paralelo, cierto tipo de aplicaciones se pueden beneficiar de una mayor integración del microprocesador y la tarjeta gráfica. Por ejemplo, la generación de imágenes tridimensionales o el procesado de imágenes fotográficas se pueden realizar en menor tiempo gracias a este nuevo diseño.

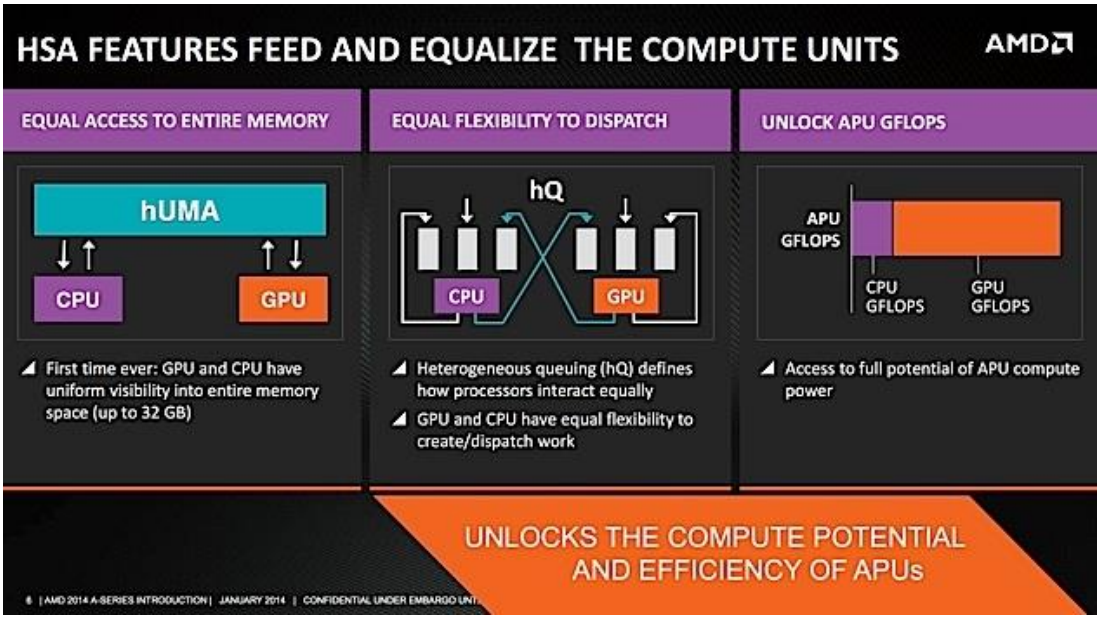

Figura 8: Características de las APUs de AMD.

La última APU desarrollada por AMD es la denominada Kaveri, una nueva generación de chips para ordenadores de sobremesa y portátiles con cuatro núcleos de CPU y ocho núcleos de GPU. La flexibilidad de la arquitectura Kaveri permite que todos los núcleos se repartan la carga de trabajo de forma eficiente, eliminando las barreras que han existido históricamente entre el hardware integrado de procesamiento gráfico y los núcleos de procesamiento central.

La nueva línea que está desarrollando AMD es el concepto de núcleos de cálculo, procesadores cuyas capacidades de cálculo gráfico y de datos generales serán totalmente intercambiables y que lo hagan en igualdad de condiciones, más allá de su propia especialización. Un ejemplo de esto es que la GPU pueda acceder a la memoria del sistema directamente para asumir tareas de procesamiento intensivo cuando la CPU integrada no pueda hacerse cargo de ellas.

Según AMD, la APU A10-7850K[13] ofrece un rendimiento bastante parejo con el Core i5-4670K[14] de Intel usando una tarjeta gráfica AMD R9 270X[15] y los parámetros gráficos de cada videojuego al máximo.

2.2 Métodos de Programación basados en directivas

El uso de métodos de programación basados en directivas no es nuevo, existen multitud de compiladores que soportan este sistema de directivas, las cuales son utilizadas para guiar al compilador a la hora de generar código eficiente y facilitar la portabilidad de código. Uno de los métodos más destacados en directivas es el paradigma de programación paralela OpenMP, el cual genera código paralelo en sistemas de memoria compartida. OpenMP aparece en su versión para C/C++ entorno al año 2000. Trata de ser un estándar para unificar las soluciones de todos los fabricantes de sistemas de memoria compartida. En el ámbito de programación por directivas para GPU hay que destacar OpenACC, cuya primera especificación aparece en noviembre de 2011 y trata de consolidarse como estándar de programación basado en directivas para aceleradores independientemente de la plataforma, siguiendo la línea que llevo OpenMP en sus inicios.

[13] http://www.amd.com/es-es/products/processors/desktop/a-series-apu

[14] http://ark.intel.com/es-es/products/75048/Intel-Core-i5-4670K-Processor-6M-Cache-up-to-3_80-GHz

[15] http://www.amd.com/es-es/products/graphics/desktop/r9

2.2.1 OpenMP

Es un modelo de programación portable y escalable que proporciona a los programadores un API para aplicaciones paralelas en sistemas de memoria compartida. Está disponible en varias arquitecturas y permite añadir concurrencia a programas escritos en Fortran y C/C++. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno. También se puede utilizar con extensiones de OpenMP o junto con MPI[16] para clusters de computadores en sistemas de memoria distribuida.

El formato de una directiva de OpenMP [15] para C/C++ es el siguiente:

#pragma omp <directiva> [cláusula [, ...] ...]

En cuanto al modelo de ejecución de OpenMP, sigue el paradigma *fork-join* proveniente de los sistemas Unix, que consiste en dividir (*fork)* una tarea en N threads (hilos), uniendo (*join)* en el hilo principal los resultados que llegan de vuelta de cada thread cuando finaliza. Cuando se introduce una directiva OpenMP sobre una región de código, este bloque quedara marcado como paralelo y se incluirá una sincronización sobre él por medio de barreras a la finalización de los distintos hilos. Este comportamiento se puede alterar por medio de la directiva *nowait.*

Para cada thread se genera un identificador, que puede ser accedido en tiempo de ejecución por medio de la función omp_get_thread_num(). Por norma general, al hilo principal o master se le asigna el id (identificador) 0. Se puede controlar mediante funciones o variables de entorno como OMP_NUM_THREADS el número de hilos que se desea ejecutar en las regiones paralelas.

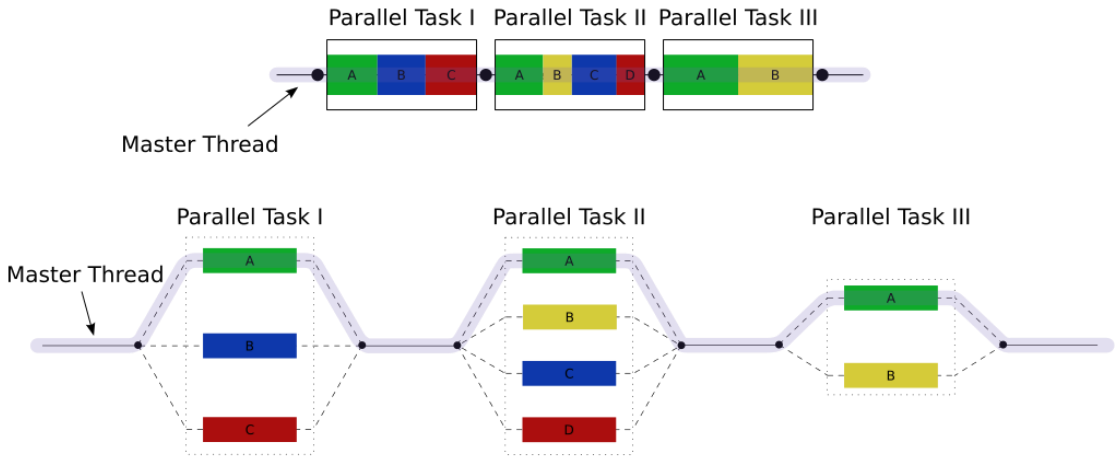

Figura 9: Modelo de ejecución en OpenMP.

[16] http://www.mcs.anl.gov/research/projects/mpi/

La directiva que define el comienzo de una sección de código paralela que será ejecutada por varios threads tiene el siguiente formato.

#pragma omp parallel [clause [[,]clause]..] new-line

{Structured block}

Las Worksharing constructs o construcciones de trabajo compartido define como se distribuye la ejecución entre los distintos hilos. Algunas de estas directivas son *for*, que indica que las iteraciones de un bucle serán ejecutadas en paralelo por diversos threads. Otra construcción que ejecuta un conjunto de bloques estructurados de código, que no son ejecutados de forma iterativa sino que cada bloque de código es ejecutado por uno de los threads se define con la directiva *sections***.** La directiva *single* obliga a que la región de código sea ejecutada por un solo thread y no obligatoriamente por el thread master. En cambio la directiva *master* si obliga a que el bloque de código sea ejecutado por el hilo principal, el master.

Se pueden anidar directivas, por ejemplo, para una región de código que contenga un bucle paralelizable se usaría la siguiente directiva, **#pragma omp parallel for**.

En cuanto al modelo de memoria en OpenMP se trata de un modelo de memoria compartida. Por lo tanto todos los threads tienen acceso para almacenar y retirar información de esta memoria. OpenMP define por medio de cláusulas de compartición de datos dos ámbitos para las variables *share* y *private*. Las variables *share* son compartidas por todos los threads sobre la variable original, por lo que hay que tener especial cuidado. Por ejemplo, en el caso de un array, controlar que no existan dependencias de datos y que no accedan varios threads a las mismas posiciones de memoria, o tener un problema de sincronización de threads lo que causaría resultados inesperados. En el caso de las variables definidas como *private*, cada thread crea una copia privada de la variable inicial pero sin inicializar, a la que solo podrá acceder el propio thread. Durante la región de código paralela cada thread referencia a su copia privada de la variable. La cláusula *firstprivate* indica que las copias privadas de las variables se inicializan con el valor de la variable original. En cambio *lastprivate* fuerza a que la variable tenga, al salir de la región privada, el valor que tendría en una ejecución secuencial. Por último la cláusula *reduction* permite hacer una reducción de los valores que hayan calculado varios threads y unirlos en una sola variable.

La última versión OpenMP 4.0 fue lanzada en marzo de 2013. Entre otras novedades esta versión incluye soporte para descargar regiones de código en dispositivos aceleradores. La región destino se crea como un thread. Por medio de la construcción *target* se crea el entorno de ejecución en el dispositivo. La directiva *map* (para mapeo de datos) acepta las clausulas *alloc* (para reservar memoria), *to* (copia el valor inicial del host), *from* (devuelve el valor modificado en el device al host) y *tofrom* (realiza copia de entrada y salida). La nueva directiva *update* permite la actualización de valores entre el host y el device. Otras

novedades son las construcciones *teams*, que definen equipos (conjuntos) de threads agrupados; y *distribute*, que permite la distribución de iteraciones de tareas entre teams. En esta versión también se ha añadido la construcción SIMD para vectorizar bucles paralelizables y bucles en serie haciendo uso de instrucciones SIMD (Single Instruction Multiple Data).

2.2.2 OpenACC

Actualmente OpenACC [16] es uno de los modelos de programación de directivas más destacados, ya que intenta consolidarse como un estándar en computación paralela para diversos aceleradores. Fue desarrollado por un grupo de compañías PGI, CAPS, NVDIA y CRAY. Su objetivo es simplificar la programación en sistemas heterogéneos CPU/GPU, facilitar la migración de código y ser un intermedio entre la paralelización automática y la paralelización manual.

Sigue la misma premisa que OpenMP, proporciona un conjunto de directivas para el compilador que indican bucles, regiones de código y regiones de datos paralelizables que serán enviadas al dispositivo acelerador en forma de kernels. Entendemos como *host* a la CPU, que será la que lleve el flujo del programa secuencial; y *device o dispositivo* al acelerador que recibirá las regiones de código que se pueden ejecutar en paralelo. Las directivas se pueden aplicar sobre programas escritos en C, C++ o Fortran.

La estructura del programa y los datos marcados por las directivas serán analizados por los compiladores de OpenACC de manera que la codificación recaerá en estos y asignará los bucles de forma que se optimice y se saque el máximo rendimiento a las capacidades hardware de los hilos y a las capacidades SIMD de los aceleradores.

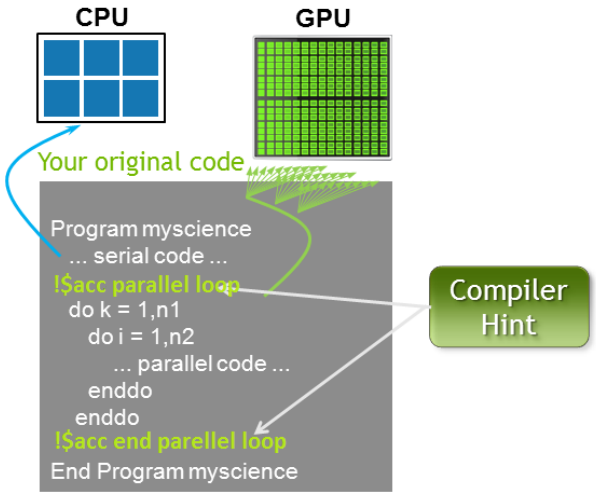

Figura 10: Modelo de ejecución en OpenACC.

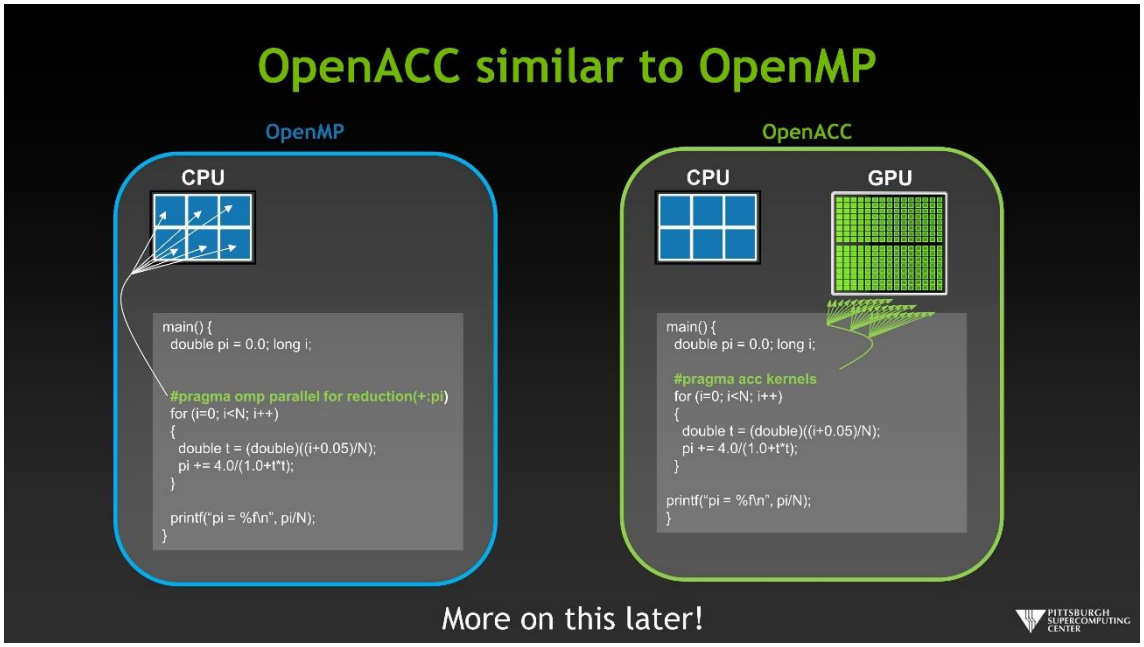

Figura 11: Modelo de ejecución en OpenMP vs OpenACC.

En cuanto al modelo de ejecución de OpenACC, sigue la misma filosofía que OpenCL o CUDA; hay una conexión entre el host y el dispositivo, el flujo de trabajo lo lleva el *host* y el *device* se encarga de ejecutar las regiones paralelas. El *host* se encarga de la reserva de memoria en el dispositivo, del envío de datos, de la descarga de código en el acelerador y del envío de los argumentos requeridos en la región paralela. También se ecargará de mantener una cola con el código que será ejecutado en el acelerador para traer de vuelta los datos y loa resultados al host.

El programador es responsable de conocer los diversos niveles de paralelismo que ofrecen los dispositivos (grano fino, grano grueso, SIMD u operaciones vectoriales). En caso de bucles sin dependencias y totalmente paralelizables, se puede utilizar paralelización de grano grueso. En cambio en el caso de que existan dependencias, hay que dividir los bucles o utilizar paralelización de grano fino o secuencial.

Existen dos directivas para definir regiones paralelas (nomenclatura para C):

- Construcción *kernel*. Mediante esta directiva indicamos al compilador que queremos generar un kernel que será enviado al dispositivo. Estamos indicando que esta región contiene código paralelizable y será el compilar el encargado de gestionar y decidir cuál es la manera más óptima de llevar a cabo la paralelización.

  #pragma acc kernels [clause [[,] clause]…] new-line
  {Región paralela}

- Construcción *parallel.* Estamos indicando al compilador que es otra región de código paralelo que será enviado al dispositivo. La diferencia es que con esta sentencia el programador puede añadir clausulas adicionales que permiten un mayor control sobre cómo se aprovechará el hardware disponible para llevar a cabo la paralelización.

  #pragma acc parallel [clause [[,] clause]…] new-line
  {Región paralela}

Este control sobre el paralelismo que permite OpenACC mediante las clausulas antes mencionadas viene dado por los conceptos de *gangs, workers* y *vector.* Estos conceptos pueden variar dependiendo del compilador y de la arquitectura. Estos conceptos, comparadoss con la arquitectura CUDA, se muestran en la tabla 2. Usando el compilador PGI[17] sería equivalente a decir que un *vector* es equivalente a un CUDA *thread;* y un *worker* a un *warp* de CUDA. Sin embargo, en el compilador de CAPS[18], esta similitud sería justo al contrario. Dependiendo del compilador, un *gang* sería un conjunto de *workers* o *vectors.* Aplicando estos conceptos a los niveles de paralelismo de los dispositivos anteriormente mencionados, el paralelismo de grano grueso es a nivel de *gang,* pudiendo ser ejecutados N gangs en el dispositivo indicándose mediante la cláusula adicional *num_gangs* de la construcción *parallel.* Siguiendo la definición de workers y vectors de CAPS, el paralelismo de grano fino es a nivel de *workers* y el paralelismo a nivel de operaciones vectoriales y SIMD de un worker es a nivel de *vectors.* Para ajustar el número de workers en el dispositivo se utiliza la cláusula *num_workers*.

CUDA	**OpenACC**
host	*host*
thread	*vector*
block	*gang*
warp	*worker*
device	*device*

Tabla 2: Equivalencia de conceptos CUDA y OpenACC según la definición del compilador PGI.

Al ejecutarse código en el *device* se lanzan uno o más gangs compuestos de uno o más workers, que a su vez pueden estar compuestos por una o más rutas vectoriales.

[17] Compilador PGI: http://www.pgroup.com/resources/accel.htm
[18] Compilador CAPS: http://www.caps-entreprise.com/products/caps-compilers

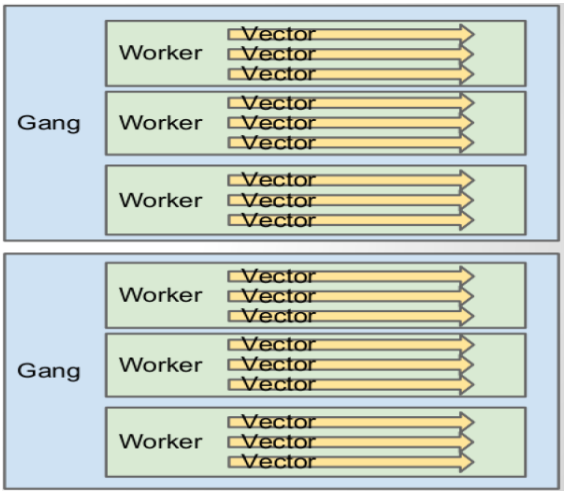

Figura 12: Relación entre gang, worker y vector de OpenACC según el compilador PGI.

La construcción *loop* se coloca antes de un bucle o un conjunto de bucles anidados. Indica al compilador que tenemos un bucle que queremos mandar a la GPU o al dispositivo acelerador que estemos usando. Dispone de distintas cláusulas que indican como se tratará este bucle en el dispositivo. La cláusula *independent* indica que no existen dependencias en el bucle y que por lo tanto puede ser paralelizado; por el contrario la cláusula *seq*, indica que el bucle será ejecutado de forma secuencial dentro del acelerador, normalmente porque contiene algún tipo de dependencia de datos. Podemos usar la cláusula *collapse(n)* para indicar al compilador que puede paralelizar los n bucles anidados que suceden a la directiva de OpenACC, esto sería equivalente a colocar un *acc loop independent* antes de cada uno de los n bucles. También se pueden aplicar a la construcción *loop* las clausulas *gang*, *worker* y *vector* para un mayor control. Por último, entre las clausulas más habituales esta *reduction*, que es empleada para realizar una operación de reducción sobre los datos resultantes del bucle.

#pragma acc loop [clause [[,] clause]…] new-line
{Bucle o conjunto de bucles anidados}

Respecto al modelo de memoria en OpenACC, hay que destacar la diferencia entre un programa que es ejecutado solo en host o solo en CPU y un programa ejecutado en un sistema heterogéneo host y device. En los sistemas con aceleradores generalmente se dispone de memorias separadas, por un lado la memoria accesible desde el host y por otro lado la memoria del dispositivo que es independiente. No pueden leer ni escribir directamente la una en la otra, por lo que son necesarias transferencias de datos entre el host y el device. Normalmente se realizan por DMA, siendo este proceso el que provoca el mayor cuello de botella en la aplicación. Por esto hay que estudiar con detenimiento que datos son necesarios transferir y en qué momento para minimizar el número de transferencias, ya que un número elevado de estas podría hacer que no fuese rentable la paralelización.

Mediante la construcción *data*, definimos una región (encerrada entre corchetes) donde los datos serán visibles por la GPU. Esta construcción también dispone de varias cláusulas que modificarán el tratamiento de los datos. Mediante la cláusula *copy(datos),* indicamos que los datos serán enviados a la GPU a la entrada de la región delimitada por la directiva *data* y devueltos al host al final de la región, en el cierre de los corchetes. Esto implica dos transferencias entre el host y el device, con la correspondiente latencia que implican este tipo de transacciones. Para evitar esto, cuando sea posible, existen las clausulas *copyin* y *copyout* para enviar datos del host al dispositivo y la operación inversa respectivamente. También disponemos de la cláusula *create* para indicar que queremos reservar memoria en la GPU, la cual estará accesible mientras dure la región delimitada por la directiva data y se liberará a la finalización de la misma. Sería similar a hacer un *cudamalloc*. Con la cláusula *present* le indicamos al compilador que los datos ya están presentes en la GPU y no necesitan ser transportados.

#pragma acc data [clause [[,] clause]…] new-line
{Región codigo}

A continuación, un pequeño fragmento de código en C (algoritmo 1) que realiza una multiplicación de matrices y muestra algunas de las construcciones explicadas durante este capítulo.

Algoritmo 1:

```
#pragma acc kernels create ( a[0:size] [0:size], b[0:size] [0:size] ) \
    copyout ( c[0:size] [0:size] )
{
        // Inicialización de matrices.
        #pragma acc loop independent
        for (i = 0; i < size; i++)
        {
                #pragma acc loop independent
                for (j = 0; j < size; j++) {
                        a[i][j] = (float)i + j;
                        b[i][j] = (float)i - j;
                        c[i][j] = 0.0f;
                }
        }

        // Calcular la multiplicación de matrices
        #pragma acc loop independent
        for (i = 0; i < size; ++i)
        {
                #pragma acc loop independent
                for (j = 0; j < size; ++j) {
                        #pragma acc loop seq
                        for (k = 0; k < size; ++k) {
                                c[i][j] += a[i][k] * b[k][j];
                        }
                }
        }
}
```

Capítulo 3. Metodología y resultados

En este capítulo, explicaremos la implementación realizada sobre el algoritmo de flujo óptico elegido, el Lucas&Kanade, cuyos motivos fueron expuestos con anterioridad. Principalmente podemos destacar el buen compromiso entre eficiencia y precisión, además de, cómo se explicó en el apartado 1.4, ser un algoritmo que ha sido objeto de numerosos estudios obteniendo buenos resultados y quedando demostrado así el porqué de su elección.

Mostraremos los detalles más relevantes de la implementación llevada a cabo y ejemplificaremos la relación entre el modelo computacional y el modelo matemático detallado en el apartado 1.3 de esta memoria. Posteriormente se explicarán las transformaciones incrementales necesarias en el código para obtener un mejor rendimiento en las dos implementaciones llevadas a cabo para su aceleración por medio de OpenMP y OpenACC.

Por último, se mostrarán los resultados obtenidos en cuanto a precisión y una comparativa de rendimiento entre las tres versiones (la serie en la CPU, la paralela en un procesador multicore y la paralela en el acelerador GPU de NVIDIA).

3.1 Implementación de L&K.

La implementación ha sido llevada a cabo en el lenguaje de programación C, por ser junto a Fortran, los lenguajes soportados en los modelos de programación de directivas (OpenMP y OpenACC). El algoritmo tendrá como entrada una secuencia de imágenes y se aplicará la técnica de Lucas&Kanade para estimar el flujo óptico entre cada par de frames consecutivos de la secuencia.

En el algoritmo 2 se detalla un pseudocódigo con el esquema general del método. Un for que recorra todas los frames de la secuencia y que encierre los siguientes elementos, dos punteros (uno que apunte al frame actual y otro al sucesor) y el conjunto de llamadas a las funciones necesarias (las cuales calculan los datos intermedios y el sistema de ecuaciones final).

En la primera fase del algoritmo se calcularán las derivadas espaciales para los ejes x e y, a continuación se calcularán las derivadas temporales y por último se calcularán los sumatorios de los productos de las derivadas, necesarios para formar las matrices necesarias para resolver el sistema (ecuaciones definidas en el punto 1.3).

Algoritmo 2:

```
Ux,Vy = metodo_Lucas_Kanade(frames, filtroX, filtroY, filtroT, Nfilas, Ncols, Nframes)
    for  z=0; z < Nframes-1; z++ do
        A0 = frames + (Nfilas*Ncols)*z
        A1 = frames + (Nfilas*Ncols)*(z+1)
        Ix = calcular_derivada_espacial_x(A0, filtroX)
        Iy = calcular_derivada_espacial_y(A0, filtroY)
        It = calcular_derivada_temporal(frames, filtroT)
        sum_Ix2 = calcular_sumatorio_vecindades(Ix2)
        sum_Iy2 = calcular_sumatorio_vecindades(Iy2)
        sum_IxIy = calcular_sumatorio_vecindades(IxIy)
        sum_IxIt = calcular_sumatorio_vecindades(IxIt)
        sum_IyIt = calcular_sumatorio_vecindades(IyIt)
        Ux,Vy = resolver_sistema_LK(sum_Ix2, sum_IxIy, sum_Iy2, sum_IxIt, sum_IyIt)
    end for
```

Para el cálculo de las derivadas espaciales se hará uso de filtros, en el estudio de precisión se utilizarán distintos tamaños para estos filtros con la intención de obtener mejores resultados. A continuación se muestra un algoritmo en pseudocódigo (algoritmo 3) que muestra la convolución llevada a cabo para obtener la derivada espacial en el eje x. Para la derivada espacial en el eje y el procedimiento será similar, pero usando un filtro vertical y cambiando las condiciones necesarias, para recorrer el filtro por filas en vez de por columnas.

Algoritmo 3:

```
Ix = calcular_derivada_espacial_x(frameAct, filtroX, Nfilas, Ncols, tamFiltro)
    for i = 0 ; i < Nfilas ; i++ do
        for j = 0 ; j  <  Ncols;  j++ do
            tmp = 0
            for k = 0;  k  <  tamFiltro; k++ do
                posicion = j + k – (tamFiltro/2)
                if posicion >= 0 and posicion < Ncols then
                    tmp += frameAct[i * Ncols + posicion] * filtroX[k]
                end if
            end for
            Ix[i * Ncols + j] = tmp
        end for
    end for
```

La salida del algoritmo Ix es la derivada espacial en el eje x; y frameAct el frame que estamos tratando. Se realiza un recorrido por filas y columnas del frame y a cada pixel se le aplica el filtroX con el tercer bucle, realizando una comprobación del filtrado en los bordes con la instrucción condicional del interior.

En el algoritmo 4 mostraremos el cálculo de la derivada temporal. Se realiza mediante una convolución sobre los k frames consecutivos correspondientes, siendo k el tamaño del filtro termporal.

Algoritmo 4:

```
It = calcular_derivada_temporal(frames, filtroT, Nfilas, Ncols, tamFiltro)
    for i = 0 ; i < Nfilas ; i++ do
        for j = 0 ; j  <  Ncols;  j++ do
            It[i * Ncols +j] = 0.0
            for k=0; k < tamFiltro; k++ do
                It[i * Ncols + j] += frames[(i * Ncols + j) + (Nfilas*Ncols*k)]*filtroT[k]
            end for
        end for
    end for
```

Para obtener los productos y los cuadrados de las derivadas necesarios para el cálculo de los sumatorios se realiza mediante una multiplicación elemento a elemento de las matrices implicadas. Por su simplicidad no es necesario especificar su algoritmo en esta memoria.

En los sumatorios de los pixeles de la vecindad, para la elección de los pesos que se asigna a cada pixel, se ha elegido el valor 1; por lo que realmente solo es necesario realizar el sumatorio de los elementos de la vecindad y asignárselo al pixel que se esté tratando. Se describe su funcionamiento en el algoritmo 5. Siendo tamBloqY y tamBloqX las dimensiones vertical y horizontal de la vecindad de pixeles.

Algoritmo 5:

```
sumaI = calcular_sumatorio_vecindad(frameAct, Nfilas, Ncols, tamBloqY, tamBloqX)
    for i = 0 ; i < Nfilas ; i++ do
        for j = 0 ; j <  Ncols; j++ do
            tmp = 0
            for k = 0; k < tamBloqY; k++ do
                for l = 0; l <  tamBloqX; l++ do
                    tmp += frameAct[(i+k-(tamBloqY/2)) * Ncols + (j+l-(tamBloX/2))]
                end for
            end for
            sumaI[i * Ncols + j] = tmp
        end for
    end for
```

Por último, el core o núcleo del algoritmo para resolver el sistema (ecuación 5, apartado 1.3) que nos proporcionará el flujo óptico, se construyen la matriz B de dimensiones 2x2 y el vector C de longitud 2 correspondientes a las ecuaciones 6 y 7 (apartado 1.3) con los valores calculados con anterioridad. Realizaremos el cálculo de la matriz inversa de B (si no se puede calcular la matriz inversa, asumiremos movimiento nulo en ese punto) y terminamos con la realización de la multiplicación de la matriz inversa de B y el vector C, dando lugar a las matrices U y V que contienen el movimiento horizontal y vertical respectivamente. En el algoritmo 6 observamos un pseudocódigo de los cálculos mencionados.

Algoritmo 6:

```
Ux, Vy = resolver_sistema_LK(sumaIx2, sumaIy2, sumaIxIy, sumaIxIt, sumaIyIt, Ncols,
Nfilas, Ncols)
    for i = 0 ; i < Nfilas ; i++ do
        for j = 0 ; j <  Ncols; j++ do
            C = [ { sumaIx2[i * Ncols + j], sumaIxIy[i * Ncols + j] },
                  { sumaIxIy[ i * Ncols + j], sumaIy2[i * Ncols + j] } ]
            B = [ − sumaIxIt[i * Ncols + j], − sumaIyIt[i * Ncols + j]
            C = calcular_inversa2x2(C)
            CB = calcular_producto_matriz_vector(C, B)
            Ux = CB[0]
            Vy = CB[1]
        end for
    end for
```

3.1.1 OpenMP. Optimizaciones y transformaciones.

La metodología para acelerar el algoritmo mediante OpenMP ha sido buscar las regiones de código que muestran un mayor paralelismo por la ausencia de dependencias de datos. En el caso de este algoritmo en particular, pueden ser aceleradas todas las construcciones explicadas en el apartado anterior. Se ha aplicado el mismo tipo de construcción OpenMP para las regiones paralelizables ya que todas siguen la misma estructura (bucles for anidados). La pragma insertada en cada construcción es la siguiente:

#pragma omp parallel for private(Conjunto variables**) firstprivate(**Conjunto variables**)**

Las regiones aceleradas por medio de esta directiva han sido el core (representado en el algoritmo 6) del Lucas&Kanade, que es la parte del código donde se resuelve el sistema; por otra parte, las construcciones que calculas las derivadas espaciales y temporales; y por último la construcción que realiza el sumatorio de la vecindad de pixeles para los productos y cuadrados de las derivadas.

Una optimización que se realizó sobre la implementación original para favorecer la paralelización fue el fisionado de bucles en las convoluciones del cálculo de las derivadas espaciales y en el cálculo del sumatorio de vecindades. El motivo para fisionar los bucles fue evitar los saltos condicionales (por ejemplo if del algoritmo 3) que existían en su interior para el control del filtrado en los bordes de la imagen, ya que como sabemos, los saltos condicionales desinhiben optimizaciones a nivel de bucle que realiza el compilador como *loop-unrolling*, *software-pipelining* etc.

Podemos mostrar, por ejemplo, en el cálculo de la derivada espacial en el eje x la forma en que se realiza el fisionado de bucles; de manera que se trata en un for independiente el filtrado en el borde izquierdo de la imagen, en otro for el cómputo central y en un tercer for el cálculo del borde derecho. En el algoritmo 7 se muestra el resultado de esta transformación en lenguaje C directamente, aplicando también el uso de directivas OpenMP.

Algoritmo 7:

```
#pragma omp parallel for private(i, j, k) \
        firstprivate(NF, NC, tamFiltro, frameAct, filtroX, Ix)
for (i = 0; i < NF; i++) {
    // Borde izquierdo
    for (j = 0; j < tamFiltro/2; j++) {
        Ix[i*NC+j] = 0.0;
        for (k = tamFiltro/2 –j; k < tamFiltro; k++)
            Ix[i*NC + j] += frameAct[i*NC + (j+k – (tamFiltro/2))] * filtroX[k];
    }
    // Centro imagen
    for (j = tamFiltro/2; j < NC – (tamFiltro/2); j++) {
        Ix[i*NC+j] = 0.0;
        for (k = 0; k < tamFiltro; k++)
            Ix[i*NC + j] += frameAct[i*NC + (j+k – (tamFiltro/2))] * filtroX[k];
    }
    // Borde derecho
    for (j = NC – (tamFiltro/2); j < NC; j++) {
        Ix[i*NC+j] = 0.0;
        for (k = 0; k < tamFiltro/2 + (NC-j); k++)
            Ix[i*NC + j] += frameAct[i*NC + (j+k – (tamFiltro/2))] * filtroX[k];
    }
}
```

3.1.2 OpenACC. Optimizaciones y transformaciones.

Para la inclusión de OpenACC en la implementación, hemos seguido el mismo procedimiento que en OpenMP, es decir añadir etiquetas en las regiones de código que presentan un mayor paralelismo, aunque realizando algunas transformaciones que o bien optimizan el rendimiento en el acelerador, o bien vienen impuestas por particularidades del compilador usado para OpenACC.

Una primera transformación respecto a la implementación en OpenMP viene dada por las altas latencias que suponen las transferencias de datos entre el host y la GPU. Como se explicó en el capítulo 2, hay que minimizar dichas transferencias, porque un uso ineficiente de las mismas puede provocar que el beneficio obtenido por la paralelización se vea lastrado. Por lo tanto, el primer objetivo es minimizar el número de transferencias; para ello declaramos una región de datos con la construcción *data* antes del bucle que recorre todos los frames de la secuencia. Además, por medio de las cláusulas de esta construcción, definimos los datos que queremos traer de vuelta de la GPU (cláusula *copyout*), que en nuestro caso son las matrices que contienen la estimación de movimiento calculada. Los datos que queremos enviar solo de ida al dispositivo (cláusula *copyin*) serán la secuencia de imagenes y los filtros necesarios para el cálculo de las derivadas espaciales. Por último, se define la memoria que se quiere reservar en la GPU (cláusula *create*) y que es necesaria para el cálculo de operaciones intermedias, como son el cálculo de derivadas y sumatorios.

Algoritmo 8:

```
#pragma acc data copyout(U[0:NF*NC], V[0:NF*NC]) \
  copyin(movie[0:NF*NC*nFrames], filtroX[0:FS], filtroY[0:FS]) \
  create(Ix[0:NF*NC], Iy[0:NF*NC], It[0:NF*NC], Ix2[0:NF*NC], Iy2[0:NF*NC], \
     sum_Ix2[0:NF*NC], sum_Iy2[0:NF*NC], IxIy[0:NF*NC], IxIt[0:NF*NC], \
     IyIt[0:NF*NC], sum_IxIy[0:NF*NC], sum_IxIt[0:NF*NC], sum_IyIt[0:NF*NC])
{
     for (z=0; z<nFrames-1; z++){
          A0 = movie + (NF*NC)*z;
          A1 = movie + (NF*NC)*(z+1);
          Ix = calcular_derivada_espacial_x(A0, filtroX);
          Iy = calcular_derivada_espacial_y(A0, filtroY);
          It = calcular_derivada_temporal(A1, A0);
          sum_Ix2 = calcular_sumatorio_vecindades(Ix2);
          sum_Iy2 = calcular_sumatorio_vecindades(Iy2);
          sum_IxIy = calcular_sumatorio_vecindades(IxIy);
          sum_IxIt = calcular_sumatorio_vecindades(IxIt);
          sum_IyIt = calcular_sumatorio_vecindades(IyIt);
          resolver_sistema_LK(sum_Ix2, sum_IxIy, sum_Iy2, sum_IxIt, sum_IyIt, U, V);
     }
}
```

En el algoritmo 8 detallamos la declaración de la región de datos y la especificación de las transferencias. También se muestra el contenido de la región, que son las llamadas que calculan el Lucas&Kanade para cada pareja de frames consecutivos de la secuencia. Nótese que han sido obviadas las funciones que calculan los cuadrados o los productos entre derivadas.

La segunda optimización relevante sobre la implementación de OpenMP afecta a las convoluciones del cálculo de derivadas y al sumatorio de vecindades. Al igual que en OpenMP, se realizó un fisionado de bucles para evitar saltos condicionales. Se ha podido comprobar que esta táctica en OpenACC no es eficiente; esto es debido a que para cada bucle fisionado que se quiere paralelizar se debe generar un *Kernel* de OpenACC. Los bucles que realizaban el cómputo de los bordes de las imágenes contenían muy poca carga computacional en comparación con los que calculan el interior, de manera que se están generando *kernels* que prácticamente no van a tener trabajo. En consecuencia, es menos eficiente generar *kernels* sin carga de trabajo que tener uno solo, aunque contenga saltos condicionales.

Algoritmo 9:

```
#pragma acc kernels present(frameAct[0:NF*NC], filterX[0:FS], Ix[0:NF*NC]) \
{
    #pragma acc loop independent
    for (i=0; i<NF; i++){
        #pragma acc loop independent
        for (j=0; j<NC; j++){
            Ix[i*NC+j] = 0.0;
            if (j<FS/2) {
                #pragma acc loop seq
                for (k=(FS/2)-j; k<FS; k++)
                    Ix[i*NC+j] += frameAct [i*NC+( j+k-FS/2)] * filterX [k];
            } else if (j>=NC- FS/2){
                #pragma acc loop seq
                for (k=0; k< FS/2+(NC-j); k++)
                    Ix[i*NC+j] += frameAct [i*NC+( j+k-FS/2)] * filterX [k];
            } else {
                #pragma acc loop seq
                for (k=0; k<FS; k++)
                    Ix[i*NC+j] += frameAct [i*NC+( j+k-FS/2] * filterX [k];
            }
        }
    }
}
```

En el algoritmo 9 se muestra la convolución del cálculo de la derivada en el eje x con la optimización del fusionado de bucles, volviendo a realizarse todo el cómputo en una construcción de bucles for anidados con saltos condicionales para tratar el filtrado en los bordes. Además de esto, se muestra un ejemplo de definición de una construcción *kernel* y de cómo se indica mediante la cláusula *present* que los datos solicitados ya se encuentran en la GPU. Se indica al compilador que los bucles son totalmente paralelizables con la directiva **#pragma acc loop independent**, de igual manera, se indica que un bucle se debe ejecutar de forma secuencial debido a alguna dependencia de datos en su interior con la directiva **#pragma acc loop seq**.

Por último, cabe mencionar dos transformaciones más que vienen dadas por restricciones del compilador usado para OpenACC. La primera es realizar un *inlining* o *inline expansion* (colocar el código en lugar de la llamada a la función) de funciones dentro de los kernel, ya que el compilador no soporta llamadas a funciones dentro de estas construcciones. Esta modificación se llevó a cabo en el kernel que resuelve el sistema de ecuaciones de LK, haciendo *inlining* de la función que calcula la inversa de una matriz y la función que realiza la multiplicación de las matrices que resuelven el flujo óptico. La segunda transformación fue, en el mismo kernel que resuelve el sistema, declarar los elementos de la matriz C 2x2 y el vector B de longitud 2 como escalares en vez de en forma matricial ya que el compilador da problemas a la hora de declarar matrices y vectores estáticos.

3.2 Resultados obtenidos. Rendimiento respecto al modelo original en C.

En este apartado presentaremos los resultados que se han obtenido a partir del conjunto de pruebas realizadas sobre la implementación lineal y sobre las implementaciones paralelas (por medio de OpenMP y OpenACC) del algoritmo de flujo óptico Lucas&Kanade. En primer lugar mostraremos un estudio sobre la precisión del algoritmo utilizando distintas configuraciones de filtros espaciales y variando el tamaño de ventana en los sumatorios de vecindades de pixeles. En una segunda fase mostraremos los resultados obtenidos en cuanto a rendimiento y en cuanto a la mejora obtenida en los modelos paralelos sobre el modelo lineal.

3.2.1 Entorno de trabajo.

Para la realización de pruebas sobre la implementación desarrollada en OpenMP se ha utilizado un sistema con dos procesador Intel Xeon E5530[19] de 4 cores cada uno, a 2.40GHz, con 8MB de caché y tecnología Hyperthreading; realizando las pruebas en configuraciones de 2, 4, 8 y 16 cores y haciendo uso del compilador gcc en su versión 4.7.2. En cuanto a la implementación realizada de OpenACC, el acelerador usado ha sido una GPU Tesla K20c de NVIDIA[20], la cual cuenta con un procesador gráfico Kepler GK110, con 2496 CUDA cores, 5GB GDDR5 de memoria y un rendimiento en punto flotante de 3.52Tflops; y haciendo uso del compilador para OpenACC pgcc de PGI, en su versión 14.6.

3.2.2 Resultados de precisión.

A continuación mostraremos los resultados obtenidos en cuanto a precisión para la implementación en C del algoritmo Lucas&Kanade. Para medir el error se ha utilizado la métrica de Barron (detallada en el punto 1.1.2) sobre los estímulos sintéticos "Diverging Tree" y "Translating Tree" (apartado 1.1.1).

Realizamos el estudio de precisión usando distintas configuraciones, variando el tamaño de los filtros espaciales para los tamaños 3, 5, 7 y 9 y probando cada uno de estos filtros sobre una ventana de tamaños 5, 9, 15, 19, 26 y 30 para realizar los sumatorios de la vecindad de cada pixel. Además, estudiaremos la precisión para diferentes densidades de cálculo, es decir, mediremos el error para la imagen completa (densidad 100%), mediremos el error para la imagen eliminando el tamaño del filtro espacial en los bordes (por lo que para una resolución de 150x150 con un filtro de tamaño 5, tendremos una densidad del 93% según el siguiente cálculo ($150-5^{\wedge 2}/150^{\wedge 2}=0.93$) y por último mediremos el error eliminando el tamaño de la ventana en los bordes (por ejemplo para una resolución de 150x150 con un tamaño de ventana de 15, la densidad obtenida es del 81%).

[19] http://ark.intel.com/es-es/products/37103/Intel-Xeon-Processor-E5530-8M-Cache-2_40-GHz-5_86-GTs-Intel-QPI

[20] http://www.nvidia.com/object/tesla-servers.html

El motivo para realizar las mediciones en distintas densidades es la falta de información en los bordes de las imágenes para realizar la estimación de flujo óptico, lo que dará lugar en estas regiones a las estimaciones más imprecisas. Cuanto más nos alejemos de estas zonas hacia el interior de la imagen para medir el error global, obtendremos resultados más precisos pero como contrapartida se compromete la densidad. Por todo ello buscaremos en este estudio un equilibrio entre precisión y densidad. A partir de ahora hablaremos de error sin corrección (para referirnos a una densidad del 100%, por lo que consideramos todos los bordes), error con corrección en filtro (eliminamos el tamaño de filtro de los bordes de la imagen) y error con corrección en bloque (donde eliminamos el tamaño de ventana en los bordes de la imagen).

El error calculado se obtiene como media del error por pixel entre el flujo óptico estimado y el real dado por el *ground truth* de la secuencia. Dado que el error se obtiene por pixel, se calcula un error medio para toda la imagen, obteniendo posteriormente una media de los errores de estimación obtenidos para cada frame de la secuencia completa. Por lo tanto, los resultados mostrados a continuación son un error promedio de la estimación de movimiento en toda la secuencia de imágenes.

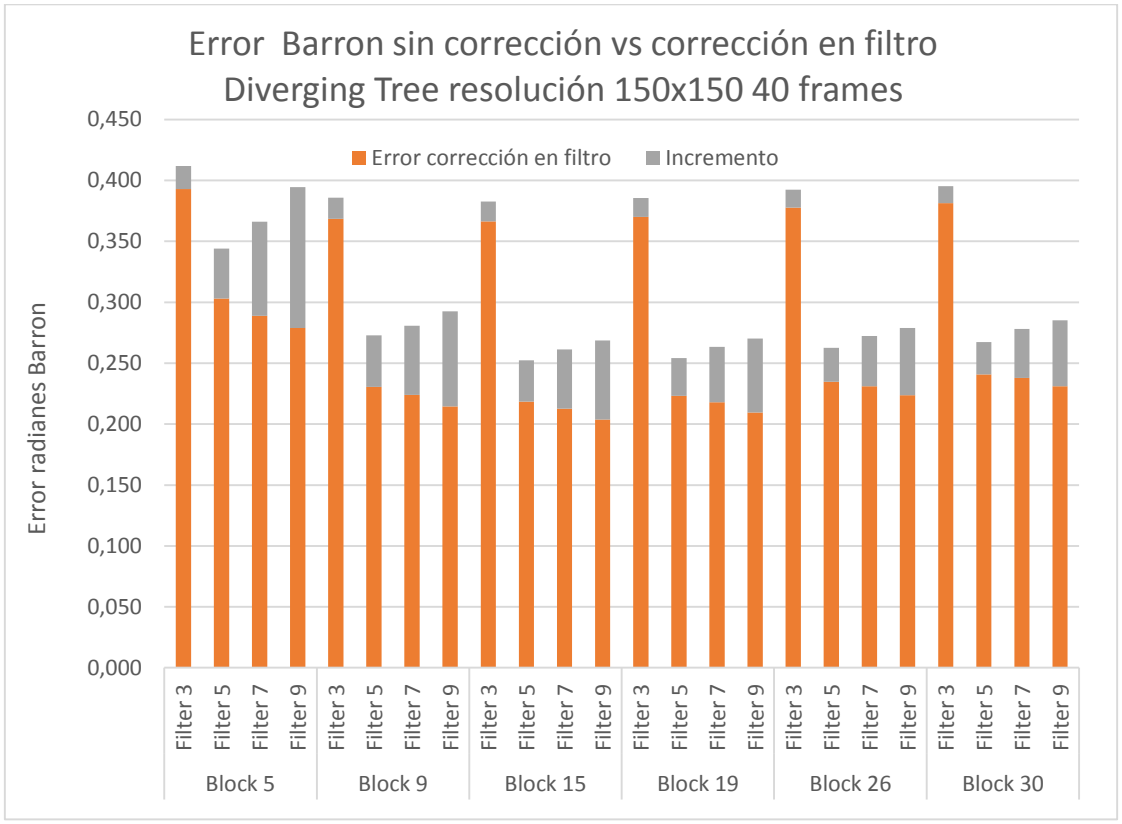

Figura 13: Gráfica del error de Barron para DivergingTree con corrección en filtro y sin corrección.

En la figura 13 se muestra un gráfico donde se aprecia el error sin corrección (barra completa) como la suma del error con corrección en filtro (barra naranja) más el incremento del error sin corrección (barra gris) en formato acumulado, (error según métrica de Barron en radianes), para la secuencia DivergingTree en distintas configuraciones (tamaños de ventana y filtro).

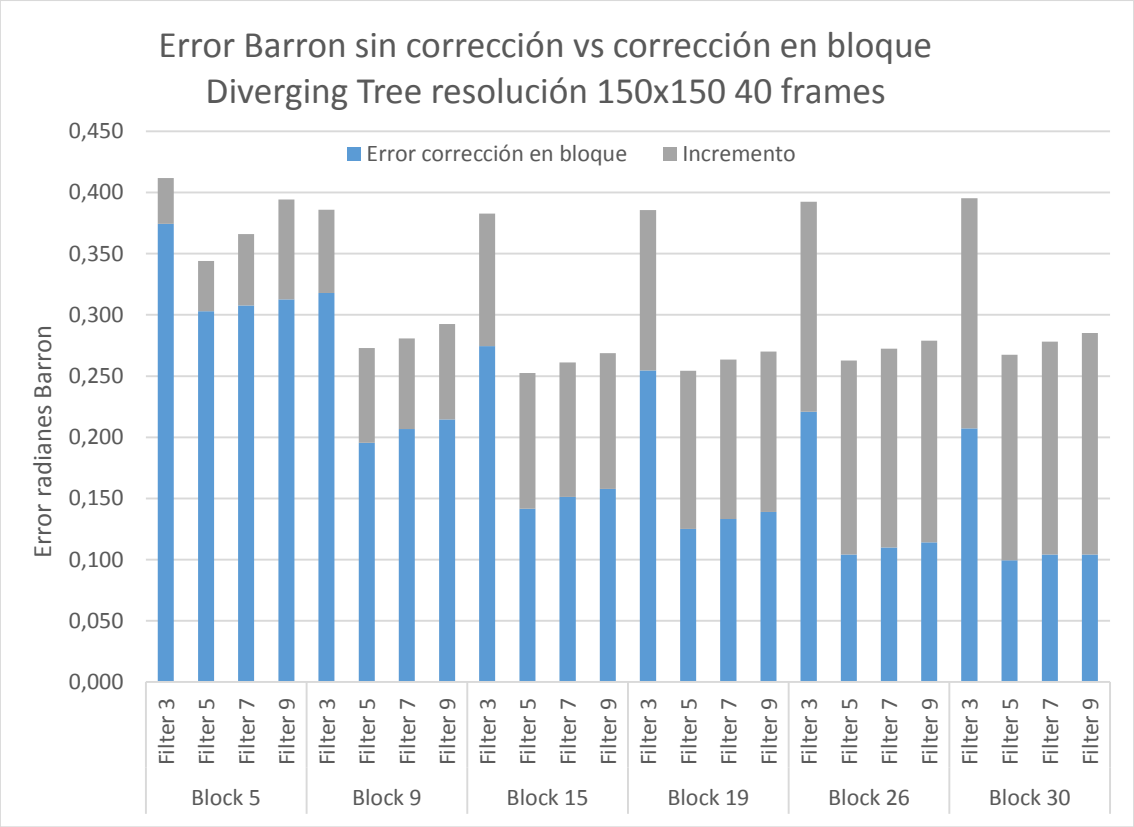

Figura 14: Gráfica del error de Barron para DivergingTree con corrección en bloque y sin corrección.

En la figura 14 tenemos un gráfico similar al anterior pero la diferencia radica en que se muestra el error con corrección en bloque (barra azul) más el incremento del error sin corrección (barra gris) representando finalmente el error sin corrección (barra completa) en formato acumulado.

Analizando las dos graficas de las figuras 13 y 14 podemos extraer la siguiente información:

Se aprecia en el caso del error sin corrección, que al disminuir el tamaño de ventana el error se va reduciendo paulatinamente encontrando su tope en la ventana de tamaño 15 ya que a partir de ésta no sigue mejorando. Respecto al comportamiento de los filtros el de tamaño 3 muestra los peores resultados, seguido del filtro 9, el filtro 7 y por último el filtro 5 que es el que mejores estimaciones produce.

Para el caso en que se calcula el error con corrección en filtro (grafico naranja) tiene un comportamiento similar en cuando a configuración, el aumentar el tamaño de bloque reduce el error llegando a su límite en el bloque 15 y en cuanto a filtros la peor estimación la realiza el filtro 3 seguido de los filtros 5, 7 y 9 siendo este último el más preciso. Además como era de esperar reduce el error en todos los casos respecto al estudio de error sin corrección.

En el error por corrección en bloque (grafico azul), se obtienen los mejores resultados pero a costa de mayores densidades. En cuanto en su comportamiento, respecto a las diferentes configuraciones observamos como el aumentar el tamaño de bloque reduce el error, yendo esta reducción de más a menos con tendencia a desaparecer para bloques superiores a los mostrados. Respecto a los filtros siguen el mismo comportamiento que en el estudio de error sin corrección, siendo el filtro 3 el más impreciso seguido del 9, 7 y 5 siendo este último el mejor.

Por último se muestra en la tabla 3 los mejores resultados para cada estudio del error de Barron junto con sus densidades, concluyendo que la corrección por filtro es la más adecuada por ser la que mejor compromiso muestra entre error y densidad de punto.

Tipo	Bloque	Filtro	Error angular	Error radianes	Densidad
Sin corrección	15	5	14.46	0.25239	100%
Corrección por filtro	15	9	11.67	0.20375	89%
Corrección por bloque	30	5	5.70	0.09945	64%

Tabla 3: Error métrica de Barron en DivergingTree para distintas correcciones junto con sus densidades.

A continuación mostramos los resultados del mismo estudio para la secuencia Translating Tree, también en resolución 150x150 con 40 frames. En la figura 15 se muestra el error sin corrección de forma acumulada (error con corrección en filtro –naranja- más incremento del error sin corrección –gris-) y en la figura 16 lo mismo pero substituyendo el error con corrección en filtro por el error con corrección en bloque (azul).

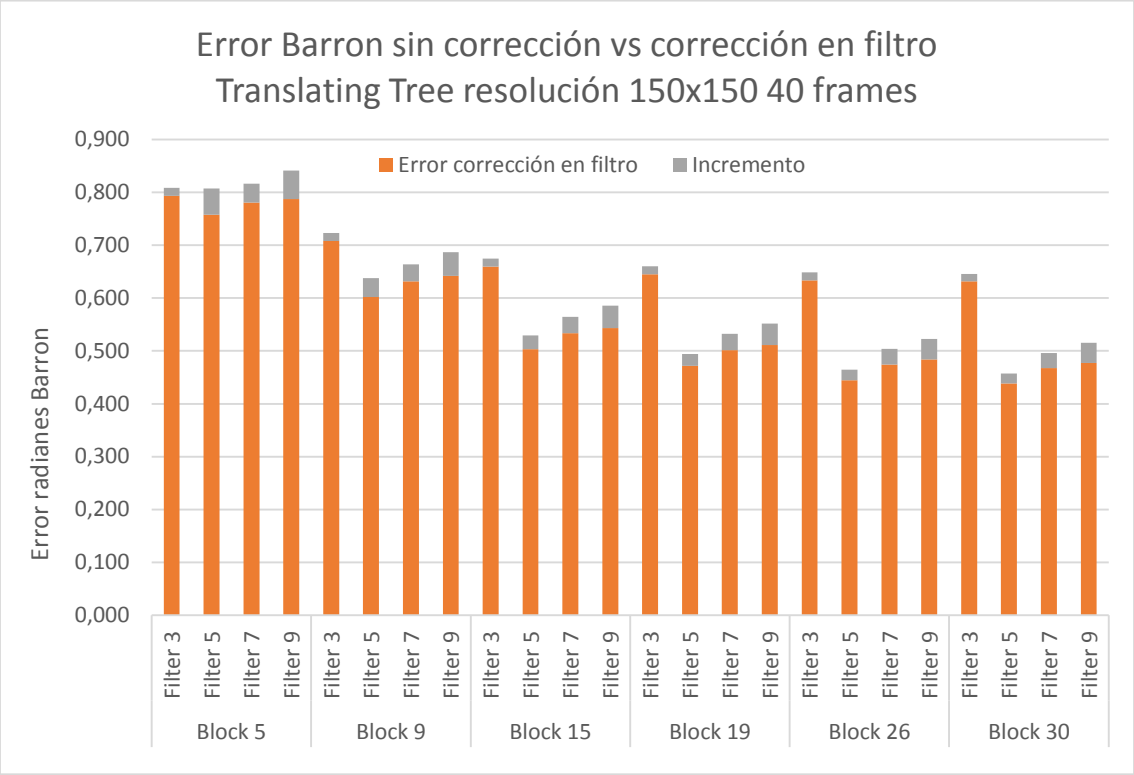

Figura 15: Gráfica del error de Barron para TranslatingTree con corrección en filtro y sin corrección.

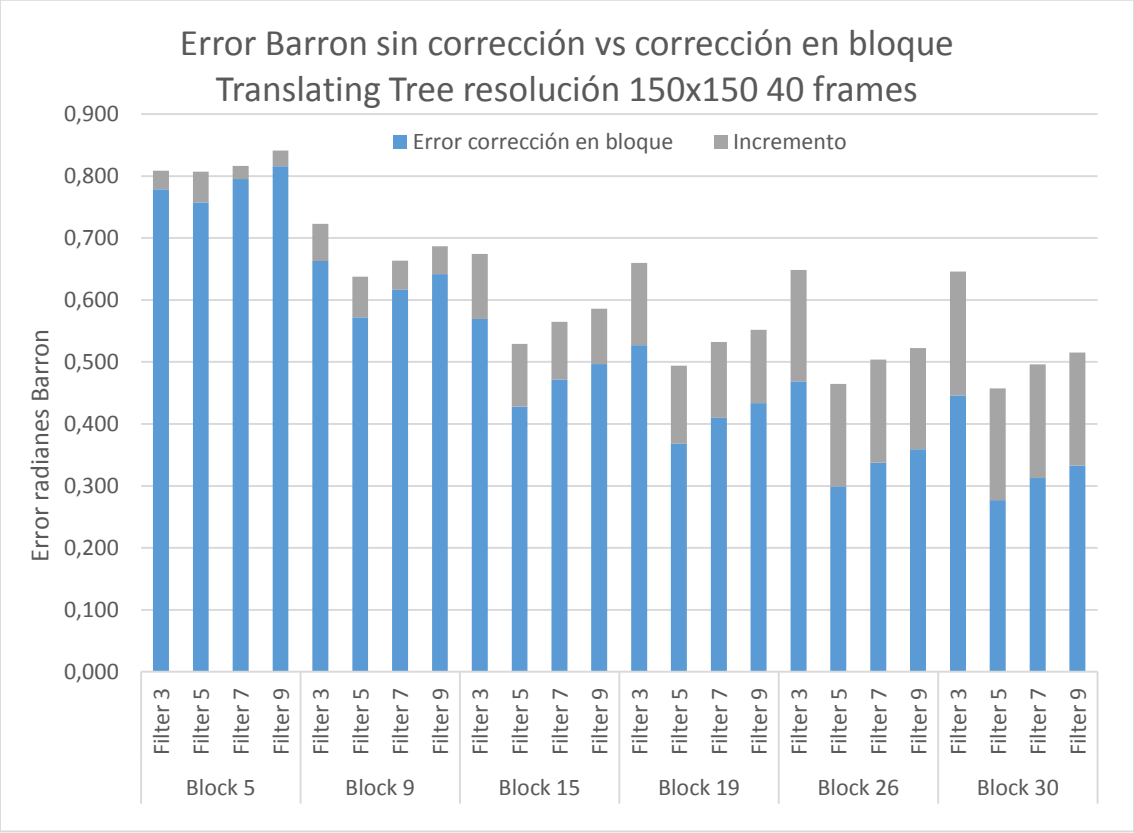

Figura 16: Gráfica del error de Barron para TranslatingTree con corrección en bloque y sin corrección.

De estas dos graficas (figuras 15 y 16), en cuanto al comportamiento de las distintas configuraciones (elección de filtro y tamaño de ventana) observamos que siguen el mismo patrón en los tres estudios de corrección (sin corrección, corrección en filtro –grafico naranja- y corrección en bloque –grafico azul-) de forma que el aumento en el tamaño de bloque disminuye el error, siendo esta reducción menor a valores mayores de bloque con tendencia a desaparecer en valores muy grandes. Respecto al comportamiento de los filtros también siguen el mismo patrón en todos los estudios, siendo el filtro 3 el más ineficiente seguido del filtro 9 y filtro 7, mostrándose en todos los casos el filtro 5 como el más preciso.

Tipo	Bloque	Filtro	Error angular	Error radianes	Densidad
Sin corrección	30	5	26.21	0.45747	100%
Corrección por filtro	30	5	25.11	0.43829	93%
Corrección por bloque	30	5	15.86	0.27683	64%

Tabla 4: Error métrica de Barron en TranslatingTree para distintas correcciones junto con sus densidades.

La tabla 4 muestra como en el estudio anterior los menores errores de Barron para cada tipo de corrección. Volviendo a mostrarse el error por corrección en filtro el que muestra un mayor compromiso entre error y densidad de punto.

Por último, se ha obtenido un promedio total de los errores obtenidos en las secuencias Diverging Tree y Translating Tree. Los resultados se muestran gráficamente en las figuras 17 y 18, mostrando la primera el error promedio sin corrección de los dos estímulos en un gráfico acumulado (formado por el error con corrección por filtro –naranja- más el incremento del error sin corrección -gris-); y mostrando la segunda el promedio entre los dos estímulos del error sin corrección (formado por el error con corrección por bloque -azul- más el incremento del error sin corrección –gris-).

Observando los gráficos de las figuras 17 y 18 podemos apreciar que en general la configuración formada por el bloque 15 y el filtro5 muestra un buen comportamiento. Corrige el error tanto si se hace una corrección por filtro como una corrección por bloque, manteniendo un buen compromiso con la densidad que se encuentra en 93% y 81% respectivamente. Consideramos por tanto esta configuración la más apropiada, ya que a partir de este tamaño de bloque aunque se mejore la precisión se ve comprometida en exceso la densidad.

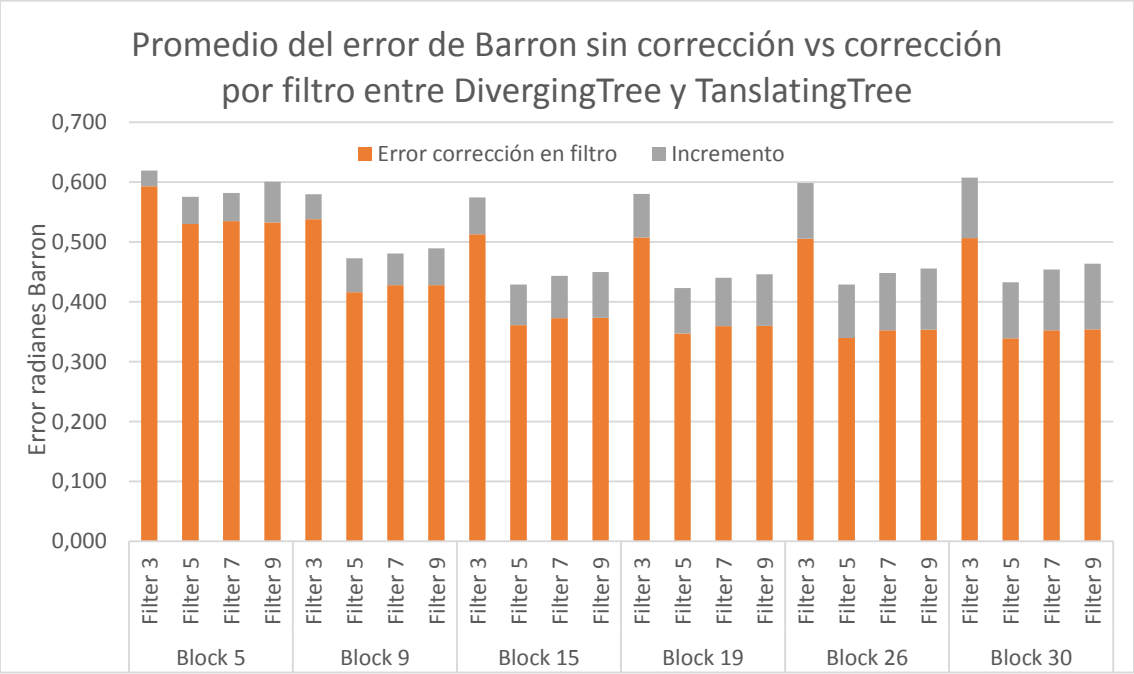

Figura 17: Gráfica del promedio del error de Barron con corrección en filtro y sin corrección entre TranslatingTree y DivergingTree.

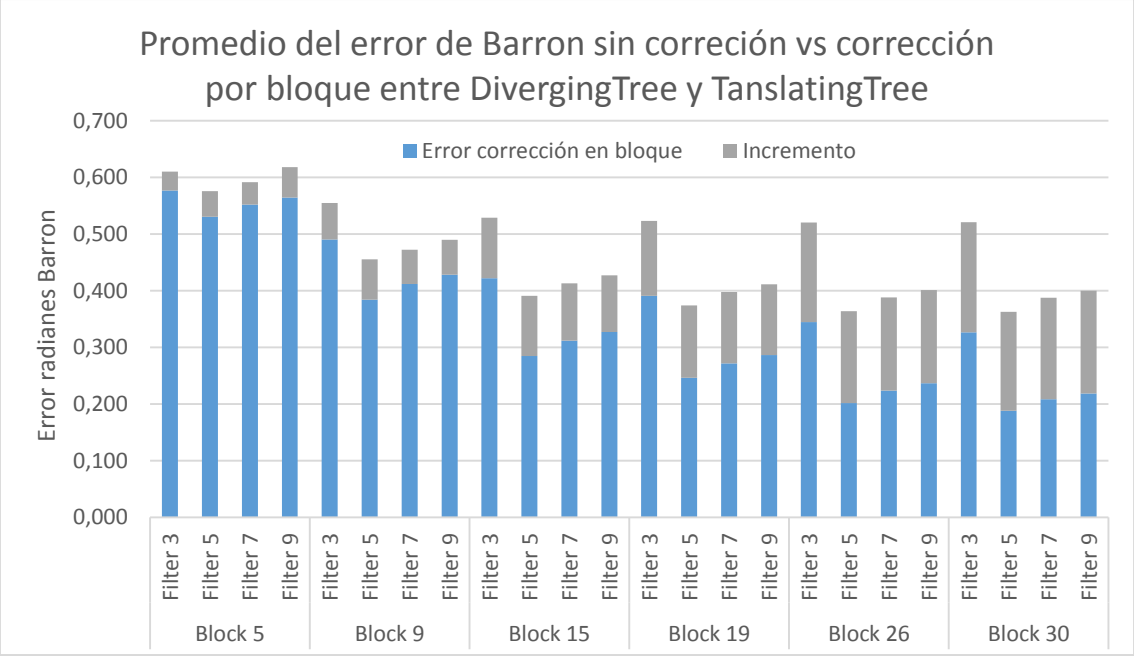

Figura 18: Gráfica del promedio del error de Barron con corrección en bloque y sin corrección entre TranslatingTree y DivergingTree.

En las figuras 19 y 20 podemos observar una representación gráfica por medio de vectores de la estimación de flujo óptico realizada por nuestro algoritmo y la representación equivalente del flujo óptico real (ground truth) para los estímulos Diverging Tree y Translating Tree respectivamente.

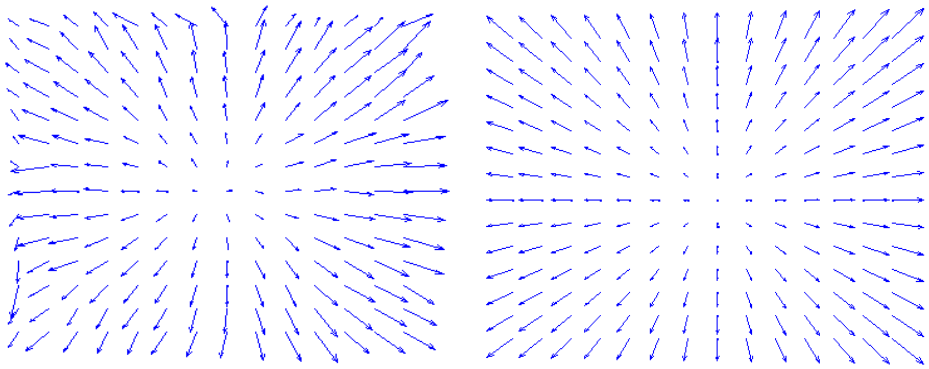

Figura 19: Flujo Óptico estimado (izquierda) y real (derecha) de Diverging Tree.

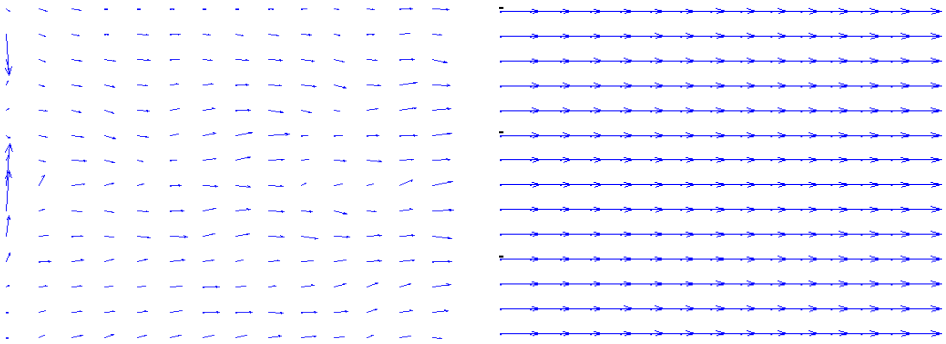

Figura 20: Flujo Óptico estimado (izquierda) y real (derecha) de Translating Tree.

Se puede observar gráficamente lo comentado al principio de este punto, sobre como los errores más considerables se encuentran en los bordes de la secuencia debido a la falta de información en estas zonas para estimar el movimiento. Algunas implementaciones toman como convenio realizar una construcción simétrica en los bordes, replicando el contenido de la imagen para que en los filtros y en las convoluciones se disponga de esa información adicional. En cualquier caso en este trabajo se ha realizado el estudio de error tratando los bordes sin corrección, con corrección en filtros y con corrección en bloque o ventana.

3.2.3 Resultados de rendimiento. Modelo original en C vs OpenMP vs OpenACC.

Para estudiar el rendimiento proporcionado por las distintas versiones de la implementación se ha optado por elegir la configuración (elección de tamaño de filtros y ventanas) con la que se obtenía la estimación de movimiento más adecuada (filtro 5 y ventana 15). Calcularemos el rendimiento en frames por segundo (fps) para distintas resoluciones de imágenes (150x150 la más baja y 2400x2400 la mayor); haciendo una comparativa de los resultados obtenidos para cada una con la implementación lineal en C, la implementación en OpenMP (paralelizando en 2, 4, 8 y 16 cores) y por último con la implementación en OpenACC. También calcularemos el rendimiento en términos de speedup observando la ganancia que proporcionan las implementaciones paralelas respecto a la implementación original en C.

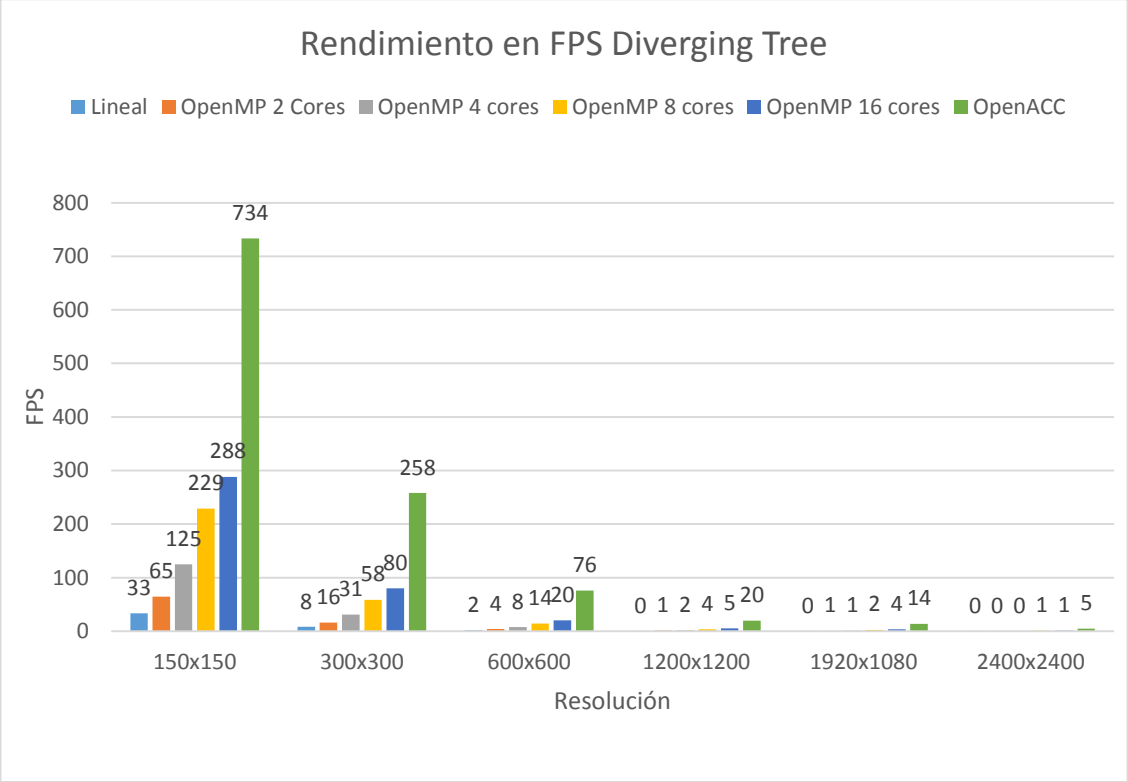

Figura 21: Gráfica del rendimiento en fps a distintas resoluciones para DivergingTree con configuración filtro 5 y ventana 15.

En el gráfico de la figura 21 podemos sacar varias conclusiones. En primer lugar observamos que el algoritmo lineal solo da resultados aceptables para resoluciones pequeñas (150x150), considerando como aceptable todo lo que se acerque al tiempo real (sobre 20 o 25fps). Con OpenMP obtenemos buenos resultados hasta una resolución de 300x300, incluso 600x600 paralelizando en 16 cores, pero a partir de esa resolución ya no obtenemos un gran beneficio. En cambio con OpenACC obtenemos tiempo real hasta una

resolución de 1200x1200, acercándose al tiempo real con 14 fps para una resolución en FullHD (1920x1080) y siendo el único que supera 1fps en una resolución de 2400x2400.

Por otra parte, si comparamos el beneficio que proporciona utilizar una implementación u otra, la clara ganadora es la de OpenACC, siendo considerable la mejora obtenida en términos de frames por segundo sobre la mejor configuración de OpenMP (16 cores). Esta mejora aumenta proporcionalmente con el tamaño de las imágenes, teniendo un factor de mejora de 2.5x para la resolución más pequeña y llegando hasta 5x para la resolución más grande (2400x2400). Si dejamos a un lado la implementación de OpenACC y nos centramos en la de OpenMP en 2, 4 y 8 cores observamos que la implementación en 8 cores tiene un factor de mejora de 7x sobre la lineal; que con la implementación de OpenMP con 4 cores tiene un factor de mejora de 2x sobre la versión de 2 cores y de 4x sobre la lineal. A su vez, la implementación de 2 cores tiene otra vez un factor de 2x sobre la lineal. Vemos por tanto que la mejora producida por OpenMP sobre la versión original sigue un crecimiento lineal y proporcional al número de cores. En cambio con OpenMP 16 cores se obtienen una mejora menor respecto a la implementación lineal, que no llega a 16x sino que se queda en 10x.

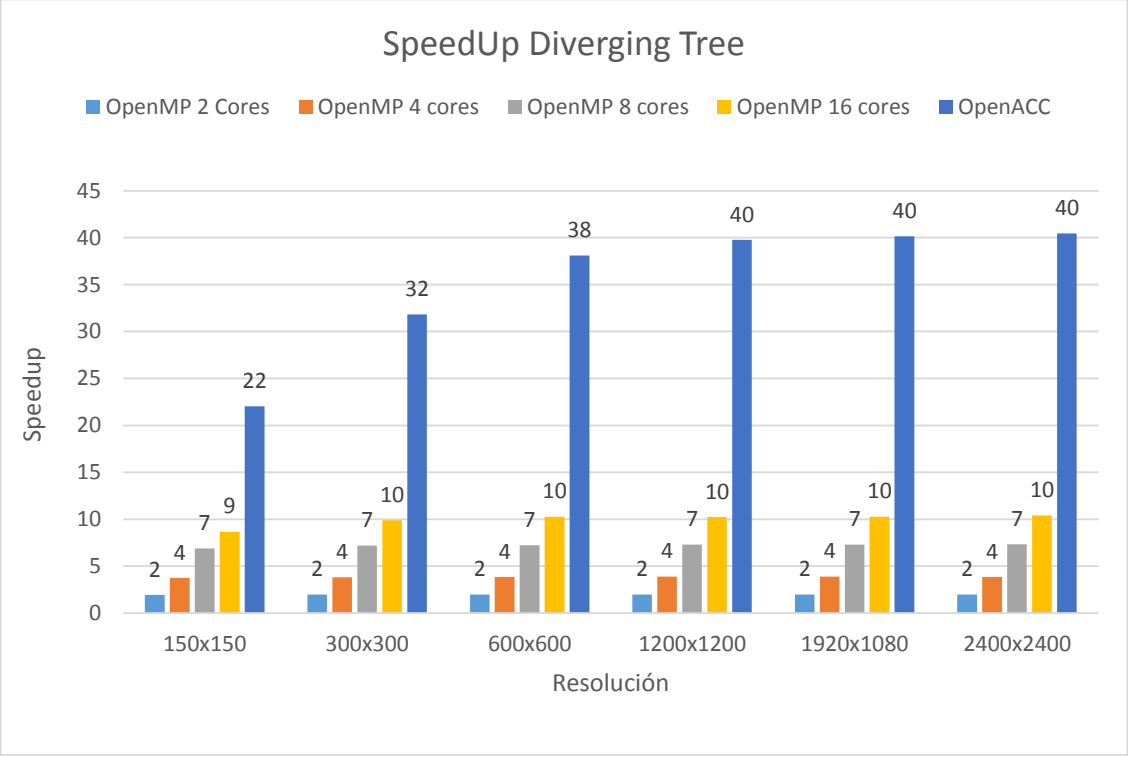

Figura 22: Gráfica del speedup sobre la versión lineal a distintas resoluciones para DivergingTree con configuración filtro 5 y ventana 15.

La figura 22 muestra el speedup en tiempos de ejecución de las diferentes implementaciones paralelas sobre la versión lineal de C. Se ve claramente que la versión de OpenMP proporciona el speedup esperado (sobre 2x para la versión 2 cores y rondando 4x y 7x para las versiones 4 cores y 8 cores respectivamente), siendo algo menor en la versión de 16 cores (speedup de 10x). Por su parte OpenACC obtiene unos speedups entre 22x y 40x.

Se observa que al aumentar la resolución de las imágenes OpenMP no proporciona una gran mejora porque ya está llegando al máximo paralelismo que puede proporcionar. En cambio, OpenACC sigue aumentando el speedup a medida que las imágenes crecen, ya que las GPUs sacan su máximo rendimiento con grandes cantidades de datos y es ahí donde explotan al máximo su paralelismo. En resoluciones de alta definición parece llegar el límite de OpenACC, donde se estabiliza el crecimiento de su speedup. Esto se aprecia más claramente en la figura 23.

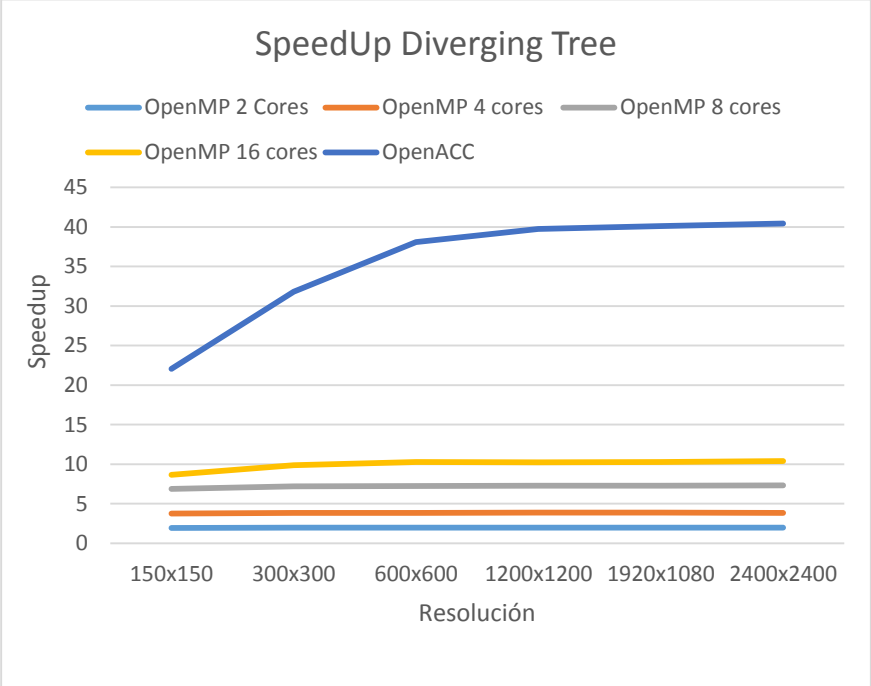

Figura 23: Gráfica de líneas para el speedup de DivergingTree con configuración filtro 5 y ventana 15.

Por último, mostramos como afectan las distintas configuraciones (elección de filtro y ventana) al speedup sobre la implementación base de C en las distintas implementaciones. En este caso los cálculos son efectuados para una resolución de 150x150 en el estímulo Diverging Tree.

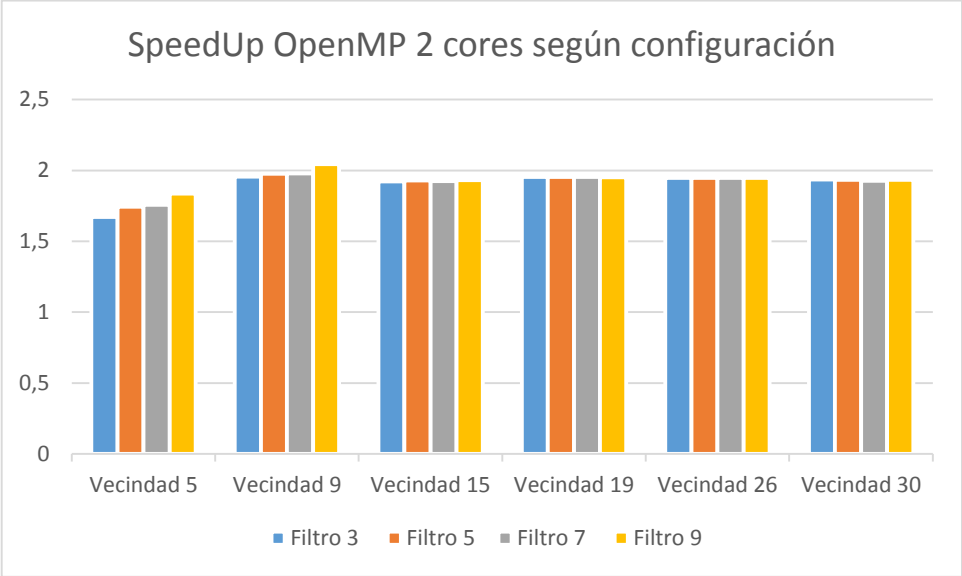

Figura 24: Gráfico del speedup de OpenMP 2 cores a resolución 150x150 de Diverging Tree.

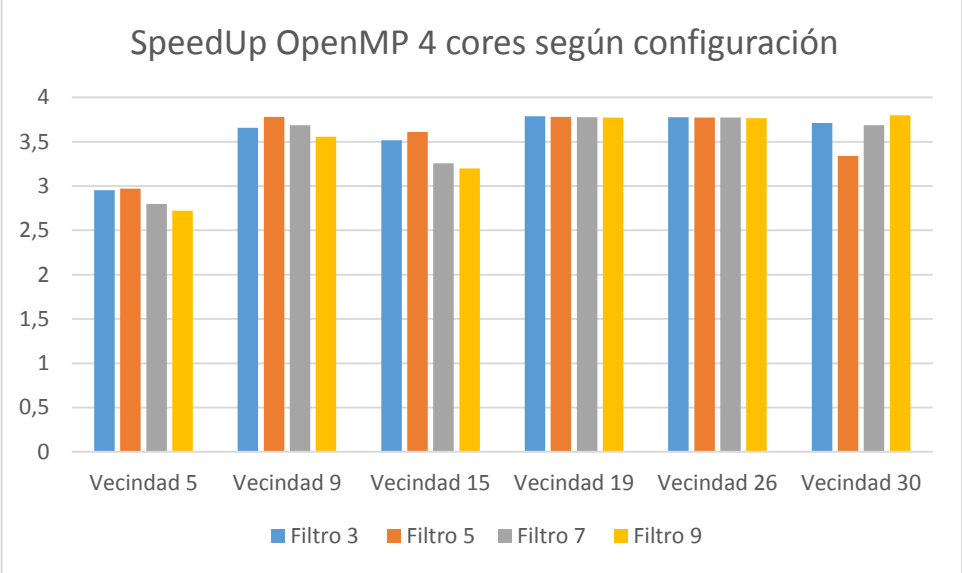

Figura 25: Gráfico del speedup de OpenMP 4 cores a resolución 150x150 de Diverging Tree.

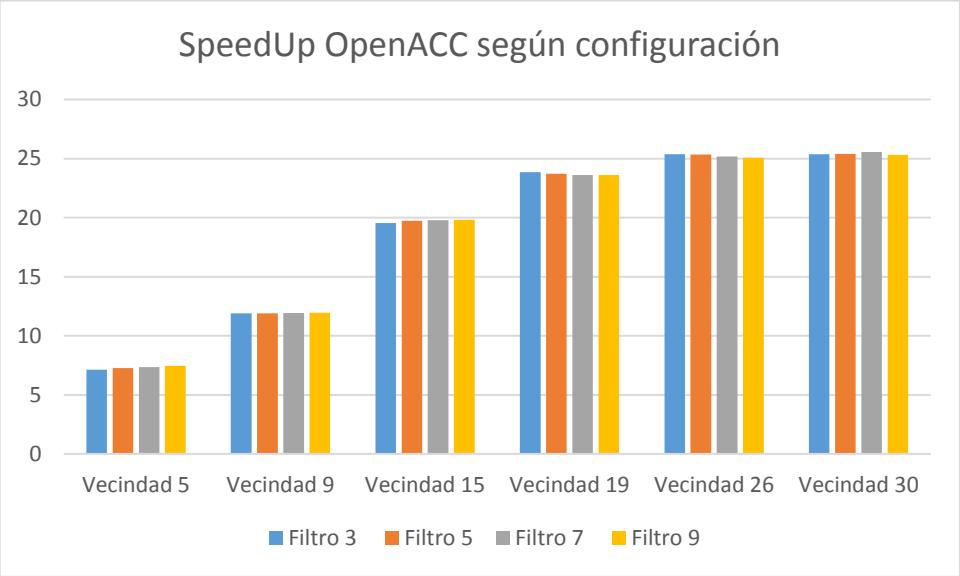

Figura 26: Gráfico del speedup de OpenACC a resolución 150x150 de Diverging Tree.

Volvemos a observar que OpenACC obtiene los mejores resultados y es el que más varía su speedup en relación al tamaño de ventana, aumentando con ésta (el speedup aumenta de 7x a 26x) y llegando a su límite en la ventana de tamaño 30; en cambio el aumento de filtro no produce una gran variación. En OpenMP observamos que el rendimiento es invariante a la configuración de filtro y ventana elegida, ya que para todas está cerca de sus límites (2x en 2 cores y de 3 a 4x en 4 cores etc). Se han omitido las gráficas de 8 y 16 cores en OpenMP ya que el resultado obtenido en cuanto a mejora de speedup con variación de filtro y ventana sigue la misma tendencia que en 2 y 4 cores, que es no verse afectado.

Todos los resultados obtenidos en el análisis de rendimiento han sido prácticamente iguales en Diverging Tree y en Translating Tree; y por este motivo no se incluyen estos últimos en la memoria. Este resultado era de esperar, ya que el rendimiento varía en función del tamaño de los datos de entrada y ambos estímulos tienen la misma resolución.

Capítulo 4. Conclusiones y trabajo futuro

4.1 Resumen y conclusiones.

Con lo expuesto a lo largo de este trabajo, en primer lugar queda clara la importancia de los problemas de estimación de movimiento en la visión por computador y en el tratamiento de video digital. Hemos podido comprobar que es un campo con diversas aplicaciones en el mundo real y de gran interés para los investigadores, vista la cantidad de proyectos y estudios que hemos encontrado documentándonos para esta memoria.

Por otra parte hemos observado el interés por realizar implementaciones más eficientes de los algoritmos que calculan el flujo óptico, ya que el tiempo requerido para resolver este tipo de problemas crece con la calidad y la resolución de las imágenes de entrada. La tendencia es tener cada vez mayor resolución en videos e imágenes, como podemos apreciar en la nueva resolución 4K que cuadriplica la resolución de la generación actual (FullHD).

En el capítulo 1 se clasificaron los modelos de algoritmos de flujo óptico y se destacó que el principal factor para decantarse entre unos u otros es buscar un equilibrio entre "accuracy & efficency" (precisión y eficiencia). Se concluyó que el algoritmo Lucas&Kanade de la familia de gradiente era un buen candidato para nuestros objetivos.

Por la necesidad de obtener una mayor eficiencia en las implementaciones y debido al paralelismo intrínseco encontrado en las imágenes (el cuál puede ser explotado en aceleradores) el siguiente paso fue conocer más acerca de éstos y ver el estado del arte actual. Hemos podido comprobar cómo en un inicio los procesadores gráficos sólo tenían la finalidad de procesar gráficos, pero su gran potencia de cálculo ha hecho que evolucionen hacia un propósito más generalista y poder aprovechar así su rendimiento en ámbitos científicos y de computación paralela. Además, actualmente la tendencia es que los aceleradores no sean sólo procesadores gráficos, sino también sistemas heterogéneos como las APUs de AMD o el Xeon Phi de Intel.

Hemos observado que uno de los mayores impedimentos para que la utilización de aceleradores tenga una mayor aceptación por parte de programadores no especializados, es la complejidad de tener que conocer arquitecturas más específicas como CUDA u OpenCL, que tienen una curva de aprendizaje lenta. Por este motivo surgen los modelos de programación por el método de directivas como el estándar OpenMP y recientemente OpenACC un prometedor estándar, pensado para explotar los aceleradores de forma independiente de la arquitectura.

Después de este trabajo creemos que OpenACC es una opción muy interesante a tener en cuenta ya que ha dado muy buenos resultados, siendo su aprendizaje asequible y pudiendo migrar un código en C de manera cómoda sin tener que realizar demasiadas modificaciones. Quizás lo más costoso haya sido conseguir tener una implementación óptima en C, pero una vez conseguida, la migración no resultó dificultosa.

Para finalizar, podemos decir que OpenACC ofrece buenos resultados, viendo el estudio realizado en el capítulo 3, donde obtenemos "speedups" de hasta 40x cuando en OpenMP lo máximo que hemos logrado está cerca de 10x. En cuanto al rendimiento en fps, OpenACC ha sido la única implementación que ha logrado resultados en tiempo real en resoluciones de hasta 1200x1200 y quedándose muy cerca en resolución FullHD (1920x1080). Puede que en implementaciones realizadas en CUDA los resultados sean más eficientes, pero si tenemos en cuenta la curva de aprendizaje, consideramos que es una seria opción a tener en cuenta.

4.2 Posibles trabajos futuros.

El estudio realizado sobre el algoritmo Lucas&Kanade y las múltiples aplicaciones de la estimación de movimiento permiten pensar en continuar varias líneas de investigación sobre él, como por ejemplo variaciones de este modelo que proporcionen más información sobre el flujo óptico o realicen estimaciones más precisas. Entre las líneas futuras que podrían plantearse deberían tenerse en cuenta las siguientes ideas:

Un posible trabajo futuro sería realizar una implementación sobre el modelo piramidal del algoritmo Lucas&Kanade. En este tipo de implementación, el flujo óptico es estimado por primera vez sobre la imagen de menor resolución, luego, dicha imagen va creciendo en un factor de 2 hasta llegar a su tamaño de adquisición, es decir, el de mayor resolución posible.

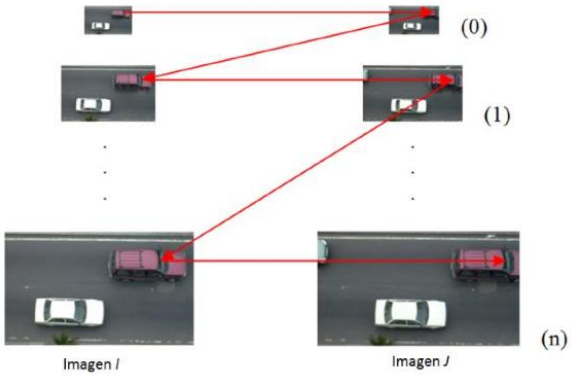

Figura 27: Forma piramidal del algoritmo de Lucas&Kanade.

El modelo piramidal estima primero el flujo óptico sobre el nivel 0 (Figura 27) entre las imágenes I y J, encontrando el desplazamiento d0, de un punto específico, sobre una imagen de baja resolución y una ventana pequeña. Luego ubica en la imagen I del nivel 1 el punto encontrado anteriormente y estima de nuevo el flujo óptico entre I y J para el nivel 1, encontrando el desplazamiento d1 sobre la misma ventana. Estos pasos se repiten hasta alcanzar el nivel n, obteniendo el desplazamiento total del punto.

Otra vía de investigación podría ser centrarse en la estimación del movimiento en estéreo mediante el Lucas&Kanade. El problema que trata de resolver una implementación en estéreo es el de determinar la estructura tridimensional de una escena teniendo dos o más imágenes obtenidas desde varias perspectivas. Visión en estéreo, trata de usar dos cámaras y obtener dos imágenes sobre un mismo punto en una escena concreta, como se muestra en la figura 28. Lo que queremos conseguir es tener información sobre la profundidad de los objetos de la imagen y no solo sobre el movimiento en 2D (planos horizontal y vertical) como tenemos con el modelo básico de Lucas&Kanade. Para llevarlo a cabo, habría que ejecutar en paralelo el algoritmo desde dos dispositivos, cada uno tomaría la imagen desde un punto, por lo que podría calcularse la distancia y la profundidad.

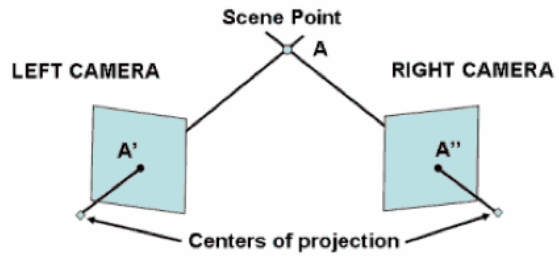

Figura 28: Obtención de una imagen tridimensional del "Scene Point" a partir de dos imágenes tomadas con dos cámaras.

Otra posible aplicación sería llevar estos algoritmos a plataformas móviles, ya que los smartphones y tablets de última generación, que están en pleno auge, contienen sistemas para capturar video e imágenes. Además, en la actualidad la tendencia también es que cuenten con dispositivos gráficos como los NVIDIA Tegra[21], por lo que se convierten en buenos candidatos para abrir un campo de estudio sobre estas plataformas.

[21] http://www.nvidia.com/object/tegra.html

Chapter 4. Conclusions and future work

4.1 Summary and conclusions.

The statements in this work clarify, firstly, the importance of the problems on motion estimation in computer vision and in digital video processing. We have verified that this is a field with many applications in real world and of great interest to researchers, after seeing the great number of projects and studies we have found while we were documenting for this memory.

Moreover, we have seen interest in making more efficient implementations of algorithms that calculate the optical flow, since the time required to solve such problems increases with the quality and resolution of the input images. The trend is to have more and more a higher resolution in videos and images, as we can appreciate in the new resolution 4K which quadruples the resolution of the current generation FullHD.

In Chapter 1, models of optical flow algorithms were classified, and it was also emphasized that the main factor for deciding among them is to find a balance between accuracy and efficiency. It was concluded that Lucas&Kanade algorithm from gradient-based family was a good candidate for our purposes.

Because of the need on achieving more efficiency in implementations and due to the intrinsic parallelism found in the images (which can be exploited on accelerators) the next step was to learn more about them and consider the current trends. We have checked how, initially, the graphic processors only had the purpose of processing graphics, but his high computing power made them evolve into a more general purpose, being able to exploit its performance in scientific fields and parallel computing. Furthermore, the current trend is that accelerators are not just graphic processors, but also heterogeneous systems as APUs from AMD or Intel Xeon Phi from Intel.

We have noticed that one of the main obstacles so that the use of accelerators has wider acceptance by unskilled programmers, is the complexity of having to know more specific architectures like CUDA or OpenCL, that have a long learning curve. For this reason the models of programming by directives method as the standard OpenMP and recently a hopeful standard OpenACC, designed to exploit accelerators independently of the architecture.

After doing this work, we believe that OpenACC is a very interesting option to consider, since it has produced good results, being affordable its learning and being able to migrate C code in a comfortable way without making too many modifications. Perhaps, the most difficult part has been to have an optimal implementation on C, but once achieved, migration was not difficult.

Finally, we can established that OpenACC provides good results, seeing the study made in Chapter 3, where we obtained "speedups" of up to 40x when in OpenMP the maximum we achieved is close to 10x. In terms of performance in fps, OpenACC has been the only implementation that has achieved results in real time in resolutions of up to 1200x1200, and being so close in FullHD resolution (1920x1080). Maybe in implementations developed in CUDA, the results are more efficient, but if we consider the learning curve, we come to the conclusion that it is a serious option to consider.

4.2. Possible future work.

The study on Lucas&Kanade algorithm and multiple applications of motion estimation allow us to think about carrying out several lines of investigation on it, such as variations of this model that provide more information on optical flow or can perform more accurate estimations. Among the future lines that could be considered the following ideas should be taken into account:

A possible future work would be to perform an implementation of the pyramid model of Lucas&Kanade algorithm. In this type of implementation, optical flow is estimated for the first time on the lower resolution image, then, the image grows by a factor of 2 to reach its acquisition size, that is, the highest resolution possible.

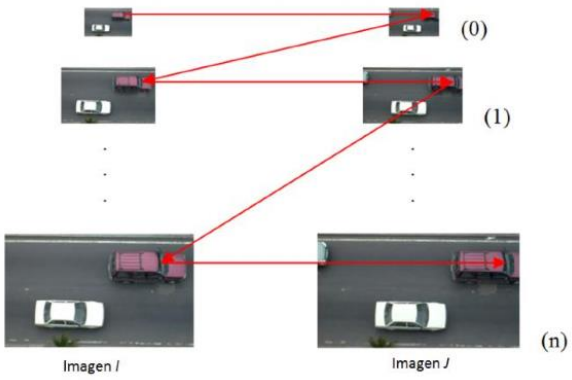

Figure 27: Pyramid form of the Lucas&Kanade algorithm.

The pyramid model estimates first the optical flow on the level 0 (Figure 27) between the images I and J, finding the d0 displacement, on a specific point, in a low-resolution image and a small window. Then it places in the image I on the level 1, the point previously found and estimates again the optical flow between I and J for the level 1, finding the displacement d1 in the same window. These steps are repeated until the level n will be reached, obtaining the total displacement of the point.

Another line of research could be to focus on the motion estimation in stereo trough Lucas&Kanade. The problem that tries to solve an implementation in stereo is to determine the three dimensional structure of a scene with two or more images taken from different perspectives. Stereo vision tries to use two cameras and to obtain two images on the same point in a particular scene, as it is shown in Figure 28. What we want to achieve is having the information about the depth of the objects in the image and not only about the 2D movement (horizontal and vertical planes) as we have with the basic model of Lucas&Kanade. To carry it out, the algorithm should be run in parallel from both devices, each one of them would take the image from a different point, which could compute the distance and depth.

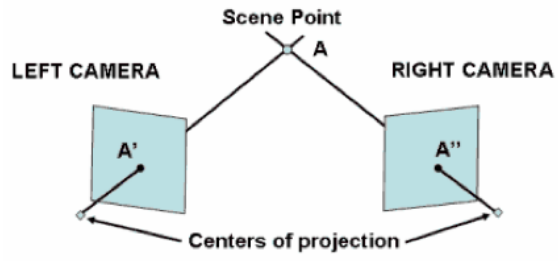

Figure 28: Getting a three-dimensional image of the "Scene Point" from two images taken with two cameras.

Another possible application would be export these algorithms to mobile platforms, as latest generation smartphones tablets, that are booming, contain systems to capture video and images. Furthermore, nowadays, the tendency is for them to have graphic devices as NVIDIA Tegra, so that they become good candidates to open a field study on these platforms.

Aportación individual al proyecto

Nelson Martín Vieites.

La aportación al proyecto ha sido conjunta, no se ha hecho una diferenciación clara de tareas con excepción de la parte de documentación, en la cual cada componente del grupo se ha especializado más en un área, pudiendose decir que mi aportación individual está presente en todos los ámbitos del proyecto. A continuación, se describe dicha aportación detalladamente, que a grandes rasgos se puede divider en dos partes; documentación y experimentación, las cuales a su vez contienen diferentes etapas.

Podemos diferenciar dos etapas en el proceso de documentación, la inicial fue situarse en el contexto de la estimación de movimiento, el flujo óptico y el algoritmo a utilizar. La segunda etapa se centra en el estudio de aceleradores, programación paralela y metodologías de programación por el método de directivas. En cuanto a la parte de experimentación podemos diferenciar cuatro subapartados. Los dos primeros se llevaron a cabo de forma paralela ya que eran complementarios, siendo estos el desarrollo en C del algoritmo y la toma de contacto con Octave[22] para el tratamiento de imágenes y visualización de resultados; llegando en último lugar a las etapas de paralelización por medio de OpenMP y OpenACC con sus respectivos estudios y toma de resultados.

La primera parte del proyecto fue documentarse sobre el flujo óptico, los tipos de algoritmos de estimación de movimiento y sus respectivas familias, profundizar en los modelos de gradiente y en el algoritmo Lucas&Kanade ya que fue el método elegido para implementar en el proyecto. En esta primera etapa también fue necesario documentarse sobre los estímulos sintéticos que se iban a utilizar, las métricas de error disponibles para medir la precisión de la estimación y realizar un estado del arte de implementaciones realizadas sobre aceleradores gráficos del algoritmo elegido (Lucas&Kanade). También, una parte del tiempo fue destinada a conocer Octave y sus comandos básicos para la manipulación de imágenes y ficheros, para poder representar gráficamente los resultados. Gran parte de toda esta documentación inicial nos fue facilitada por nuestros tutores del proyecto, ademas, otra fuente de consulta en este comienzo vino dada por la tesis doctoral de Fermín Ayuso [17].

En esta primera etapa del proyecto, una vez que ya se habían adquirido los conocimientos esenciales para empezar a trabajar sobre el algoritmo, se comenzó con los dos primeros subapartados de la fase de experimentación: la implementación en lenguaje C del algoritmo de Lucas&Kanade y la experimentación en Octave para poder visualizar gráficamente los resultados.

[22] http://www.gnu.org/software/octave/

Una vez finalizada una implementación básica en C del algoritmo, y habiendo adquirido los conocimientos básicos en lo referente a la estimación de movimiento, se pasó a la segunda fase del proyecto. Esta fase comenzó con la documentación sobre el tema de los aceleradores gráficos y con comprender el concepto de programación paralela, para luego pasar a profundizar en los dos modelos de programación basados en directivas usados en el proyecto, OpenMP y OpenACC. Después de estudiar los conceptos básicos de estos modelos, con el fin de comenzar con la migración del código original dado que los dos tienen conceptos similares, se decidió que cada componente profundizara más por separado en las particularidades de cada uno de ellos, centrándome yo más en el estudio de OpenACC.

Por último, llegó la parte de experimentación y recolección de resultados sobre las implementaciones paralelas y el estudio de precisión sobre la implementación base de C. En el estudio de precisión se analizó en un primer momento como afectaban las distintas configuraciones (elección de filtro y ventana), estudiándose en segundo lugar como podían reducirse estos errores aplicando correcciones y manteniendo un compromiso entre error y densidad de punto. Para el estudio de rendimiento en las implementaciones paralelas, primero se realizaron las optimizaciones pertinentes seguidas de la migración a OpenMP; y en último lugar se realizó el mismo proceso sobre OpenACC. También se realizaron pruebas utilizando varios compiladores para OpenACC, aunque finalmente el que mejor funcionó fue el compilador elegido, el pgcc de PGI. Para la realización del proyecto hemos tenido a nuestra disposición todos los medios y el hardware necesario en los laboratorios del departamento de Arquitectura de Computadores y Automática de la facultad de Físicas.

Una vez que se terminó con la parte práctica y de documentación del proyecto, se procedió a la realización de la presente memoria, donde se ha seguido el siguiente proceso: recopilar el contenido, recuperar la documentación consultada durante el curso, sintetizar los resultados obtenidos en la fase de experimentación, y por último, la redacción y posterior revisión del documento.

Jorge Collado García.

La temática del trabajo nos ha obligado a trabajar juntos desde el comienzo del mismo hasta final. Únicamente durante la parte de documentación, nos hemos dividido especializándonos cada uno más en un área distinta. Podemos dividir el desarrollo del proyecto en dos grandes apartados claramente distintos donde hemos tenido que dedicarle tiempo a la documentación y experimentación por partes iguales. Estos dos grandes apartados serían por una parte el desarrollo del propio algoritmo Lucas&Kanade en C y el estudio centrado en GPUs y programación paralela.

Para la primera parte del proyecto hemos tenido que documentarnos sobre el flujo óptico, los distintos modelos de gradiente, tipos de algoritmos ya existentes para la estimación de movimiento y, por supuesto, el concepto del algoritmo Lucas&Kanade. En esta etapa del proyecto, hemos tenido que documentarnos acerca de otros conceptos no menos importantes para el desarrollo del trabajo como son los estímulos sintéticos, su uso y las métricas de error existentes para medir la precisión de la estimación del movimiento. Destacar también que una parte de nuestro tiempo dedicado a la documentación fue para conocer los comandos básicos del Octave para poder representar gráficamente los resultados de nuestro algoritmo. Gran parte de toda esta documentación inicial nos fue facilitada por nuestros tutores de proyecto, otra fuente de consulta en este comienzo vino dada por la tesis doctoral de Fermín Ayuso [17].

Respecto al primer apartado de desarrollo del proyecto, podemos dividir nuestra experimentación en dos etapas distintas:

- Implementación del algoritmo propuesto, Lukas&Kanade, en C.
- Experimentación, en esta etapa procedimos a probar el algoritmo con diferentes estímulos sintéticos para estudiar y hacer una comparativa de resultados obtenidos.

Una vez terminado el desarrollo y la experimentación con el algoritmo, pasamos a la segunda parte de nuestro proyecto. Comenzamos este apartado estudiando los conceptos básicos de programación paralela y los dos estándares de programación basados en directivas que hemos usado a usar para nuestro trabajo de fin de grado (OpenMP y OpenACC). Posteriormente nos dedicamos a migrar el código original del algoritmo que obtuvimos en la primera etapa de nuestro proyecto. Al ser los dos estándares muy similares, decidimos que cada uno profundizara más en uno para hacer más llevadero el trabajo. Por mi parte comentar que mi tarea fue la de conocer en mayor profundidad el estándar OpenMP y mi compañero se dedicó al estudio profundo del otro estándar que teníamos, OpenACC.

En resumen, cada uno profundizó más en el estudio del estándar que había elegido, no obstante a la hora de completar el código, fue de manera conjunta ya que muchas de las

sentencias que teníamos que introducir eran las mismas pero especiales para cada estándar diferente.

Por último, llegamos a la nueva etapa de experimentación y obtención de resultados sobre estas dos implementaciones con sentencias propias de OpenMP y OpenACC. Para comenzar, hemos tenido que realizar optimizaciones necesarias para que el código migrado a OpenMP y OpenACC sea correctamente eficiente y acorde con lo que nosotros perseguíamos. Para probar los algoritmos hemos tenido a nuestra disposición el hardware necesario en los laboratorios del departamento de Arquitectura de Computadores y Automática de la Facultad de Ciencias Físicas de la Universidad Complutense de Madrid.

Finalizada ya la parte práctica del proyecto y con resultados ya obtenidos con una y otra implementación pasamos al desarrollo de la memoria presente. El proceso seguido ha sido el de recabar toda la información que fuimos anotando durante todo el curso y recopilar todos los resultados obtenidos en los distintos hardwares. Llegados a este punto pasamos a tratar esos resultados con el fin de reflejar en esta memoria los datos obtenidos para obtener esta novedosa comparativa. Por otra parte hemos tenido que sintetizar toda la información acerca de la cual nos documentamos a lo largo del curso y por último, redactarlo todo de manera clara y concisa.

Referencias

[1] J.L Barron, D.J. Fleet, and S.S. Beauchemin. Performance of optical flow techniques. INTERNATIONAL JOURNAL OF COMPUTER VISION, 12:43-77, 1994.

[2] B. Lucas & T. Kanade: An iterative image registration technique with an applications to stereo vision. Proc DARPA Image Understanding Workshop, pp. 121-130, 1984.

[3] S. Baker and I. Matthews. Lucas-Kanade 20 Years On: A Unifying Framework. International Journal of Computer Vision, 56(3):221-255, 2004.

[4] H. Liu , T.H. Hong , M. Herman , T. Camus and R. Chellappa: Accuracy vs. Efficiency Trade-offs in Optical Flow Algorithms. Computer Vision and Image Understanding. Vol.72, Issue 3 (Dec) pp. 271 – 286, 1998.

[5] B. McCane, K. Novins, D. Crannitch and B. Galvin: On Benchmarking Optical Flow. Computer Vision and Image Understanding. Vol. 84, pp 126–143, 2001.

[6] J. Díaz. Sistema de visión bio-inspirado multi-modal. Arquitectura de procesamiento de movimiento y visión estéreo de altas prestaciones. PhD thesis, Universidad de Granada, 2006.

[7] Julien Marzat, Yann Dumortier, Andr_e Ducrot, et al. Real-time dense and accurate parallel optical ow using CUDA. In 7th International Conference WSCG, 2009.

[8] Bernardt Duvenhage, J. P. Delport, and Jason de Villiers. Implementation of the Lucas-Kanade image registration algorithm on a GPU for 3d computational platform stabilisation. In Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, AFRIGRAPH '10, pages 83:90, New York, NY, USA, 2010. ACM.

[9] J. Ohmura, A. Egashira, S. Satoh, T. Miyoshi, H. Irie, and T. Yoshinaga. Multi-GPU Acceleration of Optical Flow Computation in Visual Functional Simulation. In Networking and Computing (ICNC), 2011 Second International Conference on, pages 228:234, 30 2011-dec. 2 2011.

[10] NVIDIA Corporation. CUDA: Compute Unified Device Architecture. [on line] http://developer.nvidia.com/object/cuda.html, December 2012.

[11] OpenCL [on line] http://www.khronos.org/opencl/.

[12] John L Hennessy and David A Patterson. Computer architecture: a quantitative approach. Elsevier, 2012.

[13] NVIDIA Corporation. Whitepaper: NVIDIA's Next Generation. CUDA Compute Architecture: Kepler GK110, 2012.

[14] Intel. Intel Core i7-3900 Desktop Processor Series. [on line] http://download.intel.com/support/processors/corei7/sb/core_i7-3900_d.pdf, 2013.

[15] OpenMP Application Program Interface. Version 4.0. [on line] http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013.

[16] The OpenACC Application Programming Interface. Version 2.0. [on line] http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, June 2013.

[17] F.Ayuso. Aceleración de algoritmos bioinspirados para estimación de movimiento en hardware paralelo. PhD thesis, Universidad Complutense de Madrid, 2013.